合肥學院
HEFEI UNIVERSITY

Distributed Computing
Lesson 7: Text Encoding

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://www.it-weise.de

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

Overview

- How can we deal with text?
- How can we know that a sequence of bits stands for “汤卫思” and not for “Ölüberschussländer”?

Character Encoding

- Different languages have different characters

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language (for German, we can e.g., replace some less important characters with "ä" and "ß"...)

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)
- These approaches are insufficient for other languages

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)
- These approaches are insufficient for other languages
- Universal Coded Character Set (UCS) [3] and Unicode [4–6]

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)
- These approaches are insufficient for other languages
- Universal Coded Character Set (UCS) [3] and Unicode [4–6]
- Encoded as `UTF-7`, `UTF-8` [7] (compatible to `ASCII`), `UTF-16`, and `UTF-32`

Character Encoding

- Different languages have different characters
- Originally, storage of text data was mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)
- These approaches are insufficient for other languages
- Universal Coded Character Set (UCS) [3] and Unicode [4–6]
- Encoded as `UTF-7`, `UTF-8` [7] (compatible to `ASCII`), `UTF-16`, and `UTF-32`
- When sending text data, we need to make sure to use the right encoding!

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation
- Input

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation
- Input:
  - `Reader`s read one or multiple *unicode characters*

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation
- Input:
  - `Reader`s read one or multiple *unicode characters*
  - `InputStreamReader`s are readers which take their data from a byte-based input stream

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation
- Input:
  - `Reader`s read one or multiple *unicode characters*
  - `InputStreamReader`s are readers which take their data from a byte-based input stream
- Output

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation
- Input:
  - `Reader`s read one or multiple *unicode characters*
  - `InputStreamReader`s are readers which take their data from a byte-based input stream
- Output:
  - `Writer`s write one or multiple unicode characters

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation
- Input:
  - `Reader`s read one or multiple *unicode characters*
  - `InputStreamReader`s are readers which take their data from a byte-based input stream
- Output:
  - `Writer`s write one or multiple unicode characters
  - `OutputStreamWriter`s are writers which store all characters written to them in a specified encoding into byte-based output streams

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation
- Input:
  - `Reader`s read one or multiple *unicode characters*
  - `InputStreamReader`s are readers which take their data from a byte-based input stream
- Output:
  - `Writer`s write one or multiple unicode characters
  - `OutputStreamWriter`s are writers which store all characters written to them in a specified encoding into byte-based output streams
- TCP sockets: plug the `InputStreamReader` and `OutputStreamWriter`s directly into the streams that the socket offers to us

Character Encoding in Java

- In Java, we can use the more general Character Stream API [8] to deal with data conversation
- Input:
  - `Reader`s read one or multiple *unicode characters*
  - `InputStreamReader`s are readers which take their data from a byte-based input stream
- Output:
  - `Writer`s write one or multiple unicode characters
  - `OutputStreamWriter`s are writers which store all characters written to them in a specified encoding into byte-based output streams
- TCP sockets: plug the `InputStreamReader` and `OutputStreamWriter`s directly into the streams that the socket offers to us
- UDP sockets: create the packets in memory

Character Encoding in C

- Usually determined at compile-time

Character Encoding in C

- Usually determined at compile-time
- Different types for different characters and encodings: `char`, `TCHAR`, `wchar_t`, ...

Listing: MinHTTPServer.java Minimum HTTP [9–11] Server / Java

```
import java.io.BufferedReader;      import java.io.File;        import java.io.FileInputStream;   import java.io.InputStreamReader;
import java.io.OutputStreamWriter;  import java.io.PrintWriter; import java.net.ServerSocket;     import java.net.Socket;

public class MinHTTPServer { //this is a minimum web server; see lesson 07 coming later
  public static final void main(final String[] args) {
    ServerSocket server;   Socket client;   BufferedReader  br;   int  i;   byte[]    bs;
    PrintWriter  pw;       String s;        FileInputStream fis;  File f;   Throwable x;

    try {
      server = new ServerSocket(9995); //create server socket (1 + 2)

      for (;;) {
        client = server.accept(); //wait for and accept incoming connections 3)

        br = new BufferedReader(new InputStreamReader(client.getInputStream())); // read character data
        pw = new PrintWriter(new OutputStreamWriter(client.getOutputStream(), "ISO_8859-1")); //chose the right encoding! [9, 10]

        process: { //4 + 3)
          x = null;
          try {
            while ((s = br.readLine()) != null) { //read text from connection line-by-line until end
              if (s.startsWith("GET␣")) {  // try to find the GET command in the HTTP request [9, 10]
                f   = new File(s.substring(4, s.indexOf('␣', 4)).replace('/', File.separatorChar)); //in a very crude way, extract the
                    requested path from that command
                bs  = new byte[(int) (f.length())];  //allocate a buffer of the right size
                fis = new FileInputStream(f);        //open the file
                i   = fis.read(bs);                  //read the complete file into memory
                fis.close();                         //close the file
                pw.write("HTTP/1.1␣200␣OK\r\n\r\n");  pw.flush(); //send "success" according to [9, 10]
                client.getOutputStream().write(bs, 0, i); //...and the file content  (4 + 3))
                break process;                            //ok, we are finished here
              }
            }
          } catch (Throwable t) { x= t; } //if request fails, remember why

          //hm, we did not find the file or had an error [9, 10]
          pw.write("HTTP/1.1␣404␣Not␣Found\r\n\r\n<html><head><title>404</title></head><body><h1>404␣-␣Not␣found</h1><pre>");
          if(x != null) { x.printStackTrace(pw); } //write the error message (notice the <pre>...</pre> wrapper)
          pw.write("</pre></body></html");         //end the html body
          pw.flush(); //and flush (4 + 3))
        }

        client.close();                                                    //4)
      }
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

HTTP Client: Java

Listing: MinHTTPClient.java HTTP Client: Java

```
import java.io.BufferedReader;       import java.io.InputStreamReader;
import java.io.OutputStreamWriter;   import java.net.Socket;

public class MinHTTPClient {//this is a minimum web client; see lesson 07 coming later

  public static final void main(final String[] args) {
    String dest, request, response;   Socket sock;
    OutputStreamWriter w;             BufferedReader r;

    dest    = "www.baidu.com";       // a random example for a Chinese host
    request = "GET /index.html HTTP/1.1\nHost: " + dest + "\n\n\n";

    try {
      sock = new Socket(dest, 80);  // web servers are usually listening at port 80
      w    = new OutputStreamWriter(sock.getOutputStream());
      w.write(request);             // write the HTTP request [9–11]
      w.flush();                    // make sure that all data has been sent
      sock.shutdownOutput();        // closing down the channel for sending data to the server

      r = new BufferedReader(new InputStreamReader(sock.getInputStream(), "UTF-8")); // Baidu uses UTF-8 encoding
                                                                                     // before they used GB2312 [2] encoding
      while ((response = r.readLine()) != null) { // read strings line-by-line until connection closed by server
        System.out.println(response);             // print to output
      }

      sock.close();
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

HTTP Client: Java 1.7

Listing: MinHTTPClientJava17.java Min HTTP Client + Try-With-Resource

```
import java.io.BufferedReader;          import java.io.InputStreamReader;
import java.io.OutputStreamWriter;      import java.net.Socket;

public class MinHTTPClientJava17 {//this is a minimum web client; see lesson 07 coming later
  public static final void main(final String[] args) {
    String dest, request, response;

    dest    = "www.baidu.com";      // a random example for a Chinese host
    request = "GET /index.html HTTP/1.1\nHost: " + dest + "\n\n\n";

    try(Socket sock = new Socket(dest, 80)) { // web servers are usually listening at port 80
      try(OutputStreamWriter w = new OutputStreamWriter(sock.getOutputStream())) {
      w.write(request);             // write the HTTP request [9–11]
      w.flush();                    // make sure that all data has been sent
      sock.shutdownOutput();        // closing down the channel for sending data to the server

      try (InputStreamReader is = new InputStreamReader(sock.getInputStream());
          BufferedReader     r  = new BufferedReader(is)) { // Baidu uses UTF-8 encoding
          while ((response = r.readLine()) != null) { // read strings line-by-line until connection closed by serve
            System.out.println(response);             // print to output
          }
        }
      }
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

Summary

- Text is a very complex variable-length data structure.
- Historically, there exist many different mappings from characters to bits and bytes.
- Unicode assigns an integer number to a character.
- UTF-8 defines how such a number can be translated to a variable-length list of bits.
- UTF-8 is now the prevalent text encoding in the internet, i.e., you should store all your text-based documents (txt, html, xml, ...) in UTF-8 encoding.

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://www.it-weise.de

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog

Bibliography

Bibliography I

1. *ISO/IEC 8859-1 – Final Text of DIS 8859-1, 8-bit Single-Byte Coded Graphic Character Sets – Part 1: Latin Alphabet No.1*, volume ISO/IEC 8859-1:1997 (E). Geneva, Switzerland: International Organization for Standardization (ISO), February 12, 1998. URL `http://std.dkuug.dk/jtc1/sc2/wg3/docs/n411.pdf`.
2. Ken Lunde. *CJKV Information Processing*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999. ISBN 0-596-51447-6 and 1-56592-224-7. URL `http://examples.oreilly.de/english_examples/cjkvinfo/AppE/gb2312.pdf`.
3. *Information Technology – Universal Coded Character Set (UCS) (ISO/IEC 10646:2011)*. Geneva, Switzerland: International Organization for Standardization (ISO), 2011.
4. USA: The Unicode Consortium Mountain View, CA and Julie D. Allen. *The Unicode Standard, Version 5.0*. Reading, MA, USA: Addison-Wesley Professional, fifth edition, 2007. ISBN 0-321-48091-0 and 978-0-321-48091-0. URL `http://books.google.de/books?id=Yn1UAAAAMAAJ`.
5. The unicode consortium, 2011. URL `http://www.unicode.org/`.
6. Jukka K. Korpela. *Unicode Explained*. Internationalize Documents, Programs, and Web Sites. Sebastopol, CA, USA: O'Reilly Media, Inc., June 28, 2006. ISBN 059610121X and 9780596101213. URL `http://books.google.de/books?id=PcWU2yxc8WkC`.
7. François Yergeau. *STD 63: UTF-8, A Transformation Format of ISO 10646*, volume 3629 of *Request for Comments (RFC)*. Network Working Group, November 2003. URL `https://tools.ietf.org/html/rfc3629`.
8. Herbert Schildt. *Java 2: A Beginner's Guide*. Essential Skills for First-Time Programmers. Maidenhead, England, UK: McGraw-Hill Ltd., 2002. ISBN 0072225130 and 9780072225136. URL `http://books.google.de/books?id=YWDJJGYaLG4C`.
9. Timothy John Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*, volume 1945 of *Request for Comments (RFC)*. Network Working Group, May 1996. URL `http://tools.ietf.org/html/rfc1945`.
10. R. Fielding, J. Gettys, Jeffrey Mogul, H. Frystyk, L. Masinter, P. Leach, and Timothy John Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, volume 2616 of *Request for Comments (RFC)*. Network Working Group, June 1999. URL `http://tools.ietf.org/html/rfc2616`.
11. David Gourley and Brian Totty. *HTTP: The Definitive Guide*. Definitive Guide. Sebastopol, CA, USA: O'Reilly Media, Inc., 2002. ISBN 1565925092 and 9781565925090. URL `http://books.google.de/books?id=qEoOl9bcV_cC`.