# Hutool Java工具类文档说明

1. **字符串处理工具类 StrUtil**

这个工具的用处类似于[Apache Commons Lang](http://commons.apache.org/)中的StringUtil，之所以使用StrUtil而不是使用StringUtil是因为前者更短，而且Str这个简写我想已经深入人心了，大家都知道是字符串的意思。常用的方法例如isBlank、isNotBlank、isEmpty、isNotEmpty这些我就不做介绍了，判断字符串是否为空，下面我说几个比较好用的功能。

**1. hasBlank、hasEmpty方法**

就是给定一些字符串，如果一旦有空的就返回true，常用于判断好多字段是否有空的（例如web表单数据）。

**这两个方法的区别是hasEmpty只判断是否为null或者空字符串（""），hasBlank则会把不可见字符也算做空，isEmpty和isBlank同理。**

**2. removePrefix、removeSuffix方法**

这两个是去掉字符串的前缀后缀的，例如去个文件名的扩展名啥。

String fileName = StrUtil.removeSuffix("pretty\_girl.jpg", ".jpg") //fileName -> pretty\_girl

还有忽略大小写的removePrefixIgnoreCase和removeSuffixIgnoreCase都比较实用。

**3. sub方法**

不得不提一下这个方法，有人说String有了subString你还写它干啥，我想说subString方法越界啥的都会报异常，你还得自己判断，难受死了，我把各种情况判断都加进来了，而且index的位置还支付负数哦，-1表示最后一个字符（这个思想来自于[Python](https://www.python.org/)，如果学过[Python](https://www.python.org/)的应该会很喜欢的），还有就是如果不小心把第一个位置和第二个位置搞反了，也会自动修正（例如想截取第4个和第2个字符之间的部分也是可以的哦~） 举个栗子

String str = "abcdefgh";

String strSub1 = StrUtil.sub(str, 2, 3); //strSub1 -> c

String strSub2 = StrUtil.sub(str, 2, -3); //strSub2 -> cde

String strSub3 = StrUtil.sub(str, 3, 2); //strSub2 -> c

**4. encode、decode方法**

好吧，我承认把String.getByte(String charsetName)方法封装在这里了，原生的String.getByte()这个方法太坑了，使用系统编码，经常会有人跳进来导致乱码问题，所以我就加了这两个方法强制指定字符集了，包了个try抛出一个运行时异常，省的我得在我业务代码里处理那个恶心的UnsupportedEncodingException。

**5. format方法**

我会告诉你这是我最引以为豪的方法吗？灵感来自slf4j，可以使用字符串模板代替字符串拼接，我也自己实现了一个，而且变量的标识符都一样，神马叫无缝兼容~~来，上栗子（吃多了上火吧……）

String template = "{}爱{}，就像老鼠爱大米";

String str = StrUtil.format(template, "我", "你"); //str -> 我爱你，就像老鼠爱大米

参数我定义成了Object类型，如果传别的类型的也可以，会自动调用toString()方法的。

**6. 定义的一些常量**

为了方便，我定义了一些比较常见的字符串常量在里面，像点、空串、换行符等等，还有HTML中的一些转移字符。

OK，就到这里，剩下的一些方法自己去源码里看吧，我注释自认为写的比文档好多了，希望多多采纳。

#### 类处理工具 ClassUtil

这个工具主要是封装了一些反射的方法，使调用更加方便。而这个类中最有用的方法是scanPackage方法，这个方法会扫描classpath下所有类，这个在Spring中是特性之一，主要为[Hulu](https://github.com/looly/hulu)框架中类扫描的一个基础。下面介绍下这个类中的方法。

##### 1.scanPackage方法

此方法唯一的参数是包的名称，返回结果为此包以及子包下所有的类。方法使用很简单，但是过程复杂一些，包扫面首先会调用 getClassPaths方法获得ClassPath，然后扫描ClassPath，如果是目录，扫描目录下的类文件，或者jar文件。如果是jar包，则直接从jar包中获取类名。这个方法的作用显而易见，就是要找出所有的类，在Spring中用于依赖注入，我在[Hulu](https://github.com/looly/hulu)中则用于找到Action类。当然，你也可以传一个ClassFilter对象，用于过滤不需要的类。

##### 2.getMethods方法

此方法同Class对象的·getMethods·方法，只不过只返回方法的名称（字符串），封装非常简单。

##### 3.getClassPaths方法

此方法是获得当前线程的ClassPath，核心是Thread.currentThread().getContextClassLoader().getResources的调用。

##### 4.getJavaClassPaths方法

此方法用于获得java的系统变量定义的ClassPath。

##### 5.parse方法。

此方法封装了强制类型转换，首先会调用对象本身的cast方法，失败则尝试是否为基本类型（int,long,double,float等），再失败则尝试日期、数字和字节流，总之这是一个包容性较好的类型转换方法，省去我们在不知道类型的情况下多次尝试的繁琐。

##### 6.parseBasic方法

此方法被parse方法调用，专门用于将字符集串转换为基本类型的对象(Integer,Double等等)。可以说这是一个一站式的转换方法，JDK的方法名太别扭了，例如你要转换成Long，你得调用Long.parseLong方法，直接Long.parse不就行了……真搞不懂，所以才有了这个方法。

##### 7.castToPrimitive方法

这个方法比较别扭，就是把例如Integer类变成int.class，貌似没啥用处，忘了哪里用了，如果你能用到，就太好了。

##### 8.getClassLoader和getContextClassLoader方法

后者只是获得当前线程的ClassLoader，前者在获取失败的时候获取ClassUtil这个类的ClassLoader。

##### 9. newInstance方法

实例化对象，封装了Class.forName(clazz).newInstance()方法。

##### 10.cloneObj方法

克隆对象。对于有些对象没有实现Cloneable接口的对象想克隆下真是费劲，例如封装Redis客户端的时候，配置对象想克隆下基本不可能，于是写了这个方法，原理是使用ObjectOutputStream复制对象流。

#### 时间工具类DateUtil

考虑到Java本身对日期时间的支持有限，并且Date和Calendar对象的并存导致各种方法使用混乱和复杂，故使用此工具类做了封装。这其中的封装主要是日期和字符串之间的转换，以及提供对日期的定位（一个月前等等）。 对于Date对象，我为了便捷，使用了一个DateTime类来代替之，继承自Date对象，主要的便利在于，覆盖了toString()方法，返回yyyy-MM-dd HH:mm:ss形式的字符串，方便在输出时的调用（例如日志记录等）。

##### 1.now、today方法

返回当前时间和日期，第一个方法返回的格式为yyyy-MM-dd HH:mm:ss，第二个返回的格式为yyyy-MM-dd

##### 2.date方法

当前时间的DateTime对象（相当于new DateTime()或者new Date()），此外还提供一个重载方法，传递long类型参数，是给定一个Unix时间戳，返回这个时间戳的时间。

##### 3.yearAndSeason方法

根据给定的Date对象返回一个年份和季节的字符串，例如20132表示2013年第二季度，同时提供一个yearAndSeasons方法，传入两个时间对象，提供这两个时间范围内的所有季节字符串。

##### 4.格式化日期

格式化日期时间，最常用的是formatDateTime、formatDate方法，返回yyyy-MM-dd HH:mm:ss和yyyy-MM-dd格式的日期字符串，如果你需要特殊格式的格式化，则使用format方法，通过传入日期的模板来返回相应的字符串。formatHttpDate则是返回HTTP头部的时间格式，不经常用。

##### 5.解析日期字符串

与字符串格式化相反的是解析日期字符串，常用的是parse方法，如果不提供日期的格式参数，则只能解析如下几种形式的日期字符串：

1. yyyy-MM-dd HH:mm:ss
2. yyyy-MM-dd
3. HH:mm:ss
4. yyyy-MM-dd HH:mm

如果你的日期格式不是这几种格式，则需要指定日期格式，对于以上格式还有专门的方法对应：

1. parseDateTime
2. parseDate
3. ParseTime

##### 6.日期偏移

getBeginTimeOfDay、getEndTimeOfDay分别获得给定日期当天的开始时间和结束时间，开始时间是00:00，结束时间是23:59。 yesterday、lastWeek、lastMouth返回昨天、上周、上个月的日期。 如果需要指定日期做偏移，则使用offsiteDay、offsiteWeek、offsiteMonth来获得指定日期偏移天、偏移周、偏移月，指定的偏移量正数向未来偏移，负数向历史偏移。 如果以上还不能满足偏移要求，则使用offsiteDate偏移制定量，其中参数calendarField为偏移的粒度大小（小时、天、月等）使用Calendar类中的常数。

##### 7.diff方法

返回两个日期的时间差，参数diffField定义了这个差的单位，单位的定义在DateUtil的常量中，例如DateUtil.SECOND\_MS表示两个日期相差的秒数。

##### 8.spendMs和spendNt方法

这两个方法主要是用于统计花费时间，例如在某个方法开头标记了了一个时间，在结尾传入这个时间，就可以计算这个方法花费的毫秒数或者纳秒数。

##### 9.toIntSecond方法

将给定日期转化为yyMMddHHmm类型字符串并转换为int类型，不常用。

### 类型转换类Conver

在Java开发中我们要面对各种各样的类型转换问题，尤其是从命令行获取的用户参数、从HttpRequest获取的Parameter等等，这些参数类型多种多样，我们怎么去转换他们呢？常用的办法是先整成String，然后调用XXX.parseXXX方法，还要承受转换失败的风险，不得不加一层try catch，这个小小的过程混迹在业务代码中会显得非常难看和臃肿，于是我们把这种类型转换的任务封装在了Conver类中。

#### 1. toStr、toInt、toLong、toDouble、toBool方法

这几个方法基本代替了JDK的XXX.parseXXX方法，传入两个参数，第一个是Object类型的被转换的值，第二个参数是默认值。这些方法做转换并不抛出异常，当转换失败或者提供的值为null时，只会返回默认值，返回的类型全部使用了包装类，方便我们需要null的情况。

#### 2. 半角转全角toSBC和全角转半角toDBC

在很多文本的统一化中这两个方法非常有用，主要对标点符号的全角半角转换。

### JDBC的ORM封装

##### 开篇

端午没事儿干扩充了下Hutool的DB部分，原来只是一个简单的SQL运行器，现在加入了方言支持，封装了增删改查，引入Session从而支持事务，可以说工程量巨大，在封装过程总我还是参考了[Jodd](http://jodd.org/)的DbOom、[Jfinal](http://www.jfinal.com/)的ActiveRecord、[Apache Commons-DbUtils](http://commons.apache.org/proper/commons-dbutils/)，吸取优点，剔除我觉得没用的，再加入些自己的想法，尽量做到简单和灵活。

##### 版本支持

由于上次已经将1.0.0版的Hutool提交到了Maven中央库，且这次也是一个重大的改进，所以把这次的更新全部放到1.1.0版本中，考虑到未完全测试，所以你可以在Github上clone下来使用，或者看下我的代码，体会下我的思想以及设计哲学（哲学这个词好有逼格……）。

##### 由来

考虑到[Hibernate](http://hibernate.org/orm/)做ORM的复杂性，它想把一个对象映射到数据库，再加上各种外键的复杂对应关系，当时学习的时候整的我焦头烂额，而且其数据库连接配置全部放在xml里，需要连接池插件去为它开发对应的插件，显然这样做太霸道了，总之这种灵活性的缺失，导致在使用[Hibernate](http://hibernate.org/orm/)的时候必须按照它指定的思路和方式走，痛苦万分啊，例如你执行一条SQL语句，查询的结果是个让人百思不得其解的列表，难用的要死。后来我便倾向于[Apache Commons-DbUtils](http://commons.apache.org/proper/commons-dbutils/)，谁让业务简单呢，没几张表，简简的看了其源码，做了些简单的改进放到我的[Hutool](https://github.com/looly/hutool)里来了，然后就看了[Jfinal](http://www.jfinal.com/)的ActiveRecord，那段时间正在看Python的一个框架[Django](https://www.djangoproject.com/)，发现其异曲同工之妙的ORM方式：将数据库表映射为一个Map，而不是一个对象。这样的灵活性大大的增加，字段也更加灵活。于是按照这个思想开始动工，封装增删改查的常用方法。

##### 对象解释

###### 1. Entity

在ORM中，我把一张表中的一条数据映射成为一个叫做Entity的类，继承自HashMap，key是字段名，value是Object类型，字段值，这样一个Entity对象就是数据库表中的一条记录，当然这个对象中还有个字段是表的名字，方便之后的操作。之后对数据库增删改查操作的对象大多是这个。

这个对象充当着两种角色，一个是数据的载体，表示一条数据，另一个就是where语句的条件，当然，Entity对象只支持 = 操作，更复杂的操作我以后再想别的办法。充当where条件时，key依旧是字段名，value是字段条件值。例如：

Entity where = Entity.create(TABLE\_NAME).set("条件1", "条件值");

表示的where语句是：

WHERE `条件1` = 条件值

当然到时候会用PreparedStatement，不会出现SQL注入。

###### 2. Table Column

这两个对象主要是描述数据库表结构的，暂时和ORM本身没啥关系，只是当你想获得一些字段信息的时候，这样来获得表结构信息：

/\*\*

\* 获得表的元数据

\*

\* @param ds 数据源

\*/

private static void getTableMetaInfo(DataSource ds) {

// 获得当前库的所有表的表名

List<String> tableNames = DbUtil.getTables(ds);

Log.info("{}", tableNames);

/\*

\* 获得表结构 表结构封装为一个表对象，里面有Column对象表示一列，列中有列名、类型、大小、是否允许为空等信息

\*/

Table table = DbUtil.getTableMeta(ds, TABLE\_NAME);

Log.info("{}", table);

}

##### 整体的架构

整体分为几部分

1. 数据源 DataSource
2. SQL执行器 SqlExecutor
3. CRUD的封装 SqlConnRunner SqlRunner
4. 支持事务的CRUD封装 Session
5. 各种结果集处理类 handler
6. 数据库的一些工具方法汇总 DbUtil

还有就是没有列出来的dialect（数据库方言），我会根据给定的DataSource、Connection等对象自动识别是什么数据库，然后使用不同的方言构造SQL语句，暂时支持的数据库有MySQL、Oracle、SqlLite3，当然如果识别失败会用ANSI SQL，这样遇到不支持的数据，可以搞定大部分方法。

下面解释下：

###### 1. 数据源

无论是JDNI还是数据库连接池，最终给用户的都是一个DataSource，那么，我执行SQL的连接直接从数据源里拿就可以，至于数据源是JNDI还是哪种连接池我是不管的，这样大大提高了灵活性。在ds包中，我还自己封装了SimpleDataSource对象，这是不用数据库连接池的数据源，连接直接问DriverManager中拿，完全是JDBC原生的数据库连接获取操作，当然这个类由于没有连接池，仅供测试或打开关闭连接非常少的场合使用。当然，我还提供了一个DruidDS把数据库配置以及连接池配置放在配置文件里，更加方便。

/\*\*

\* @return 获得数据源样例方法

\*/

private static DataSource getDataSource() {

/\*

\* 获得数据源，可以使用Druid、DBCP或者C3P0数据源

\* 我封装了Druid的数据源，在classpath下放置db.setting和druid.setting文件

\* 详细格式请参考doc/db-example.setting和doc/db/example.setting

\* 如果没有druid.setting文件，使用连接池默认的参数 可以配置多个数据源，用分组隔离

\*/

DataSource ds = DruidDS.getDataSource("test");

//当然，如果你不喜欢用DruidDS类，你也可以自己去实例化连接池的数据源 具体的配置参数请参阅Druid官方文档

DruidDataSource ds2 = new DruidDataSource();

ds2.setUrl("jdbc:mysql://fedora.vmware:3306/extractor");

ds2.setUsername("root");

ds2.setPassword("123456");

ds = ds2;

return ds;

}

###### 2. SQL执行器 SqlExecutor

这是一个静态类，里面的静态方法只有两种：执行非查询的SQL语句和查询的SQL语句

/\*\*

\* SqlExecutor样例方法<br>

\* 如果你只是执行SQL语句，使用SqlExecutor类里的静态方法即可

\*

\* @param ds 数据源

\*/

private static void sqlExecutorDemo(DataSource ds) {

Connection conn = null;

try {

conn = ds.getConnection();

// 执行非查询语句，返回影响的行数

int count = SqlExecutor.execute(conn, "UPDATE " + TABLE\_NAME + " set field1 = ? where id = ?", 0, 0);

log.info("影响行数：{}", count);

// 执行非查询语句，返回自增的键，如果有多个自增键，只返回第一个

Long generatedKey = SqlExecutor.executeForGeneratedKey(conn, "UPDATE " + TABLE\_NAME + " set field1 = ? where id = ?", 0, 0);

log.info("主键：{}", generatedKey);

/\* 执行查询语句，返回实体列表，一个Entity对象表示一行的数据，Entity对象是一个继承自HashMap的对象，存储的key为字段名，value为字段值 \*/

List<Entity> entityList = SqlExecutor.query(conn, "select \* from " + TABLE\_NAME + " where param1 = ?", new EntityHandler(), "值");

log.info("{}", entityList);

} catch (SQLException e) {

Log.error(log, e, "SQL error!");

} finally {

DbUtil.close(conn);

}

}

###### 3. CRUD的封装 SqlConnRunner SqlRunner

这两个类有些相似，里面都封装了增、删、改、查、分页、个数方法，差别是SqlConnRunner需要每个方法都传Connection对象，而SqlRunner继承自SqlConnRunner，在传入DataSource会自动获取Connection对象。Demo如下：

/\*\*

\* SqlRunner是继承自SqlConnRunner的（SqlConnRunner继承自SqlExecutor），所以相应的方法也继承了下来，可以像SqlExecutor一样使用静态方法<br>

\* 当然，SqlRunner更强大的功能在于对Entity对象做CRUD，避免写SQL语句。 SqlRunner需要实例化

\*

\* SqlRunner同时提供了带Connection参数的CRUD方法，方便外部提供Connection对象而由使用者提供事务的操作

\*

\* @param ds 数据源

\*/

private static void sqlRunnerDemo(DataSource ds) {

Entity entity = Entity.create(TABLE\_NAME).set("字段1", "值").set("字段2", 2);

Entity where = Entity.create(TABLE\_NAME).set("条件1", "条件值");

try {

SqlRunner runner = SqlRunner.create(ds);

// 指定数据库方言，在此为MySQL

runner = SqlRunner.create(ds);

// 增，生成SQL为 INSERT INTO `table\_name` SET(`字段1`, `字段2`) VALUES(?,?)

runner.insert(entity);

// 删，生成SQL为 DELETE FROM `table\_name` WHERE `条件1` = ?

runner.del(where);

// 改，生成SQL为 UPDATE `table\_name` SET `字段1` = ?, `字段2` = ? WHERE `条件1` = ?

runner.update(entity, where);

// 查，生成SQL为 SELECT \* FROM `table\_name` WHERE WHERE `条件1` = ? 第一个参数为返回的字段列表，如果null则返回所有字段

List<Entity> entityList = runner.find(null, where, new EntityHandler());

log.info("{}", entityList);

// 分页，注意，ANSI SQL中不支持分页！

List<Entity> pagedEntityList = runner.page(null, where, 0, 20, new EntityHandler());

log.info("{}", pagedEntityList);

// 满足条件的结果数，生成SQL为 SELECT count(1) FROM `table\_name` WHERE WHERE `条件1` = ?

int count = runner.count(where);

log.info("count: {}", count);

} catch (SQLException e) {

Log.error(log, e, "SQL error!");

} finally {

}

}

###### 4. 支持事务的CRUD封装 Session

Session非常类似于SqlRunner，差别是Session对象中只有一个Connection，所有操作也是用这个Connection，便于事务操作，而SqlRunner每执行一个方法都要从DataSource中去要Connection。样例如下：

private static void sessionDemo(DataSource ds) {

Entity entity = Entity.create(TABLE\_NAME).set("字段1", "值").set("字段2", 2);

Entity where = Entity.create(TABLE\_NAME).set("条件1", "条件值");

Session session = Session.create(ds);

try {

session.beginTransaction();

// 增，生成SQL为 INSERT INTO `table\_name` SET(`字段1`, `字段2`) VALUES(?,?)

session.insert(entity);

// 删，生成SQL为 DELETE FROM `table\_name` WHERE `条件1` = ?

session.del(where);

// 改，生成SQL为 UPDATE `table\_name` SET `字段1` = ?, `字段2` = ? WHERE `条件1` = ?

session.update(entity, where);

// 查，生成SQL为 SELECT \* FROM `table\_name` WHERE WHERE `条件1` = ? 第一个参数为返回的字段列表，如果null则返回所有字段

List<Entity> entityList = session.find(null, where, new EntityHandler());

log.info("{}", entityList);

// 分页，注意，ANSI SQL中不支持分页！

List<Entity> pagedEntityList = session.page(null, where, 0, 20, new EntityHandler());

log.info("{}", pagedEntityList);

session.commit();

} catch (Exception e) {

session.quietRollback();

} finally {

session.close();

}

}

###### 5. 各种结果集处理类 handler

此包中有个叫做RsHandler的接口，传入ResultSet对象，返回什么则在handle方法中自己指定。 实现的类有：

1. EntityHandler 转换为Entity列表
2. NumberHandler 当使用select count(1)这类语句的时候，或者返回只有一个结果，且为数字结果的时候，用这个handler
3. SingleEntityHandler 返回一条记录的时候用这个
4. 数据库的一些工具方法汇总 DbUtil 提供一些工具方法，最常用的就是close方法了，由于JDK7才把ResultSetStatementPreparedStatementConnection这几个接口实现了Closeable接口，所以之前只能判断类型再去关闭，这样一个close方法可以关闭多个对象。

### 单例工具——Singleton

平常我们使用单例不外乎两种方式：

1. 在对象里加个静态方法getInstance()来获取。此方式可以参考 [【转】线程安全的单例模式](http://my.oschina.net/looly/blog/152865) 这篇博客，可分为饿汉和饱汉模式。
2. 通过Spring这类容器统一管理对象，用的时候去对象池中拿。Spring也可以通过配置决定懒汉或者饿汉模式

说实话我更倾向于第二种，但是Spring更对的的注入，而不是拿，于是我想做Singleton这个类，维护一个单例的池，用这个单例对象的时候直接来拿就可以，这里我用的懒汉模式。我只是想把单例的管理方式换一种思路，我希望管理单例的是一个容器工具，而不是一个大大的框架，这样能大大减少单例使用的复杂性。

#### 使用

package com.xiaoleilu.hutool.demo;

import com.xiaoleilu.hutool.Singleton;

/\*\*

\* 单例样例

\* @author loolly

\*

\*/

public class SingletonDemo {

/\*\*

\* 动物接口

\* @author loolly

\*

\*/

public static interface Animal{

public void say();

}

/\*\*

\* 狗实现

\* @author loolly

\*

\*/

public static class Dog implements Animal{

@Override

public void say() {

System.out.println("汪汪");

}

}

/\*\*

\* 猫实现

\* @author loolly

\*

\*/

public static class Cat implements Animal{

@Override

public void say() {

System.out.println("喵喵");

}

}

public static void main(String[] args) {

Animal dog = Singleton.get(Dog.class);

Animal cat = Singleton.get(Cat.class);

//单例对象每次取出为同一个对象，除非调用Singleton.destroy()或者remove方法

System.out.println(dog == Singleton.get(Dog.class)); //True

System.out.println(cat == Singleton.get(Cat.class)); //True

dog.say(); //汪汪

cat.say(); //喵喵

}

}

#### 总结

大家如果有兴趣可以看下这个类，实现非常简单，一个HashMap用于做为单例对象池，通过newInstance()实例化对象（不支持带参数的构造方法），无论取还是创建对象都是线程安全的（在单例对象数量非常庞大且单例对象实例化非常耗时时可能会出现瓶颈），考虑到在get的时候使双重检查锁，但是并不是线程安全的，故直接加了synchronized做为修饰符，欢迎在此给出建议。

### 文件工具——FileUtil

#### 简介

我想在Java工具中，文件操作应该也是使用相当频繁的，但是Java对文件的操作由于牵涉到流，所以较为繁琐，各种Stream也是眼花缭乱，因此大部分项目里的util包中我想都有一个FileUtil的类，而本类就是对众多FileUtil的总结。

#### Linux命令对应方法

这些方法都是按照Linux命令来命名的，方便熟悉Linux的用户见名知意，例如：

1. ls 返回给定目录的所有文件对象列表，路径可以是相对ClassPath路径或者绝对路径，不可以是压缩包里的路径。
2. listFileNames 则是返回指定目录下的所有文件名，支持jar等压缩包。
3. touch 创建文件，如果给定路径父目录不存在，也一同创建。
4. del 删除文件或者目录，目录下有嵌套目录或者文件会一起删除。
5. mkdir 创建目录，父目录不存在自动创建。
6. createTempFile 创建临时文件，在程序运行完毕的时候，这个文件会被删除。
7. copy 复制文件或目录，目标文件对象可以是目录，自动用原文件名，可以选择是否覆盖目标文件。
8. move 移动文件或目录，原理是先复制，再删除原文件或目录
9. isExist 文件或者目录是否存在。

#### 常用方法

##### getAbsolutePath

获得绝对路径，如果给定路劲已经是绝对路径，返回原路径，否则根据ClassPath或者给定类的相对位置获得其绝对位置

##### close

对于实现了Closeable接口的对象，可以直接调用此方法关闭，且是静默关闭，关闭出错将不会有任何调试信息。这个方法也是使用非常频繁的，例如文件流的关闭等等。

##### equals

比较两个文件是否相同

#### 文件读写

##### 写文件

1. getBufferedWriter 获得带缓存的写入对象，可以写字符串等。
2. getPrintWriter 对 getBufferedWriter的包装，可以有println等方法按照行写出。
3. getOutputStream 会的文件的写出流想对象。
4. writeString直接写字符串到文件，会覆盖之前的内容。
5. appendString 追加字符串到文本。
6. writeLines appendLines 覆盖写入和追加文本列表，每个元素都是一行。
7. writeBytes 写字节码。
8. writeStream 写流中的内容到文件里。

##### 读文件

1. getReader 获得带缓存的Reader对象。
2. readLines 按行读取文件中的数据，返回List，每一个元素都是一行文本。
3. load 按照给定的ReaderHandler对象读取文件中的数据，ReaderHandler是一个借口，实现后就可以操作Reader对象了，这个方法存在是为了避免用户手动调用close方法。
4. readString 直接读出文件中的所有文本。
5. readBytes 读字节码

#### 其他

1. isModifed 文件是否被修改过，需要传入一个时间戳，用来比对最后修改时间。
2. getExtension 获得文件的扩展名。

### Properties替代——Setting

#### 前言

对于JDK自带的Properties读取的Properties文件，对我们来说有很多限制，首先是ISO8859-1编码导致没法加中文的value和注释（用日本的那个插件在Eclipse里可以读写，放到服务器上读就费劲了），再就是不支持变量分组等功能，因此有了Setting类。

#### 由来

配置文件中使用变量这个需求由来已久，在[Spring](http://spring.io/)中PropertyPlaceholderConfigurer类就用于在ApplicationContext.xml中使用Properties文件中的变量替换。 分组的概念我第一次在Linux的rsync的/etc/rsyncd.conf配置文件中有所了解，发现特别实用具体大家可以百度。

而这两种功能后来我在[jodd](http://jodd.org/)的[Props](http://jodd.org/doc/props.html)才有所发现，它的这个配置文件扩展类十分强大，甚至支持多行等等功能，本来想直接使用，避免重复造轮子，可是发现很多特性我完全用不到，而且没有我需要的便捷功能，于是我便造了Setting这个轮子。

#### 配置文件格式example.setting

# -------------------------------------------------------------

# ----- Setting File with UTF8-----

# ----- 数据库配置文件 -----

# -------------------------------------------------------------

#中括表示一个分组，其下面的所有属性归属于这个分组，在此分组名为demo，也可以没有分组

[demo]

#自定义数据源设置文件，这个文件会针对当前分组生效，用于给当前分组配置单独的数据库连接池参数，没有则使用全局的配置

ds.setting.path = config/other.setting

#数据库驱动名，如果不指定，则会根据url自动判定

driver = com.mysql.jdbc.Driver

#JDBC url，必须

url = jdbc:mysql://fedora.vmware:3306/extractor

#用户名，必须

user = root${driver}

#密码，必须，如果密码为空，请填写 pass =

pass = 123456

配置文件可以放在任意位置，具体Setting类如何寻在在构造方法中提供了多种读取方式，具体稍后介绍。现在说下配置文件的具体格式 Setting配置文件类似于Properties文件，规则如下：

1. 注释用#开头表示，只支持单行注释，空行和无法正常被识别的键值对也会被忽略，可作为注释，但是建议显式指定注释。
2. 键值对使用key = value 表示，key和value在读取时会trim掉空格，所以不用担心空格。
3. 分组为中括号括起来的内容（例如配置文件中的[demo]），中括号以下的行都为此分组的内容，无分组相当于空字符分组，即[]。若某个key是name，分组是group，加上分组后的key相当于group.name。
4. 支持变量，默认变量命名为 ${变量名}，变量只能识别读入行的变量，例如第6行的变量在第三行无法读取，例如配置文件中的${driver}会被替换为com.mysql.jdbc.Driver，为了性能，Setting创建的时候构造方法会指定是否开启变量替换，默认不开启。

#### 代码

代码具体请见com.xiaoleilu.hutool.demo.SettingDemo

package com.xiaoleilu.hutool.demo;

import java.io.IOException;

import com.xiaoleilu.hutool.CharsetUtil;

import com.xiaoleilu.hutool.FileUtil;

import com.xiaoleilu.hutool.Setting;

/\*\*

\* Setting演示样例类

\* @author Looly

\*

\*/

public class SettingDemo {

public static void main(String[] args) throws IOException {

//--------------------------------------------- 初始化

//读取classpath下的XXX.setting，不使用变量

Setting setting = new Setting("XXX.setting");

//读取classpath下的config目录下的XXX.setting，不使用变量

setting = new Setting("config/XXX.setting");

//读取绝对路径文件/home/looly/XXX.setting（没有就创建，关于touch请查阅FileUtil）

//第二个参数为自定义的编码，请保持与Setting文件的编码一致

//第三个参数为是否使用变量，如果为true，则配置文件中的每个key都可以被之后的条目中的value引用形式为 ${key}

setting = new Setting(FileUtil.touch("/home/looly/XXX.setting"), CharsetUtil.UTF\_8, true);

//读取与SettingDemo.class文件同包下的XXX.setting

setting = new Setting("XXX.setting", SettingDemo.class, CharsetUtil.UTF\_8, true);

//--------------------------------------------- 使用

//获取key为name的值

setting.getString("name");

//获取分组为group下key为name的值

setting.getString("name", "group1");

//当获取的值为空（null或者空白字符时，包括多个空格），返回默认值

setting.getStringWithDefault("name", "默认值");

//完整的带有key、分组和默认值的获得值得方法

setting.getStringWithDefault("name", "group1", "默认值");

//如果想获得其它类型的值，可以调用相应的getXXX方法，参数相似

//有时候需要在key对应value不存在的时候（没有这项设置的时候）告知用户，故有此方法打印一个debug日志

setting.getWithLog("name");

setting.getWithLog("name", "group1");

//重新读取配置文件，可以启用一个定时器调用此方法来定时更新配置

setting.reload();

//当通过代码加入新的键值对的时候，调用store会保存到文件，但是会覆盖原来的文件，并丢失注释

setting.setSetting("name1", "value");

setting.store("/home/looly/XXX.setting");

//获得所有分组名

setting.getGroups();

//将key-value映射为对象，原理是原理是调用对象对应的setXX方法

//setting.toObject();

//设定变量名的正则表达式。

//Setting的变量替换是通过正则查找替换的，如果Setting中的变量名和其他冲突，可以改变变量的定义方式

//整个正则匹配变量名，分组1匹配key的名字

setting.setVarRegex("\\$\\{(.\*?)\\}");

}

}

#### 对Properties的简单封装Props(版本2.0.0开始提供)

对于Properties的广泛使用使我也无能为力，有时候遇到Properties文件又想方便的读写也不容易，于是对Properties做了简单的封装，提供了方便的构造方法（与Setting一致），并提供了与Setting一致的getXXX方法来扩展Properties类，Props类继承自Properties，所以可以兼容Properties类，具体不再做介绍，有兴趣可以看下[com.xiaoleilu.hutool.Props](https://github.com/looly/hutool/blob/master/src/main/java/com/xiaoleilu/hutool/Props.java)

### Slf4j Logger 的封装——Log

很多时候我们为了在类中加日志不得不写一行，而且还要去手动改XXX这个类名

private static Logger log = LoggerFactory.getLogger(XXX.class);

第二个问题，我特别喜欢这种形式：

log.info("我在XXX 改了 {} 变量", "name");

既省去了可恶的isInfoEnabled()的判断，还避免了拼接字符串，但是呀

log.error("错误消息", e);

这样就不支持了，烦躁……

写多了就觉得这种厌烦的工作实在让人无法忍受，于是就封装了下。

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Log {

/\*\*

\* 获得Logger

\* @param clazz 日志发出的类

\* @return Logger

\*/

public static Logger get(Class<?> clazz) {

return LoggerFactory.getLogger(clazz);

}

/\*\*

\* 获得Logger

\* @param name 自定义的日志发出者名称

\* @return Logger

\*/

public static Logger get(String name) {

return LoggerFactory.getLogger(name);

}

/\*\*

\* @return 获得日志，自动判定日志发出者

\*/

public static Logger get() {

StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();

return LoggerFactory.getLogger(stackTrace[2].getClassName());

}

//----------------------------------------------------------- Logger method start

//------------------------ Trace

/\*\*

\* Trace等级日志，小于debug<br>

\* 由于动态获取Logger，效率较低，建议在非频繁调用的情况下使用！！

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void trace(String format, Object... arguments) {

trace(innerGet(), format, arguments);

}

/\*\*

\* Trace等级日志，小于Debug

\* @param log 日志对象

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void trace(Logger log, String format, Object... arguments) {

log.trace(format, arguments);

}

//------------------------ debug

/\*\*

\* Debug等级日志，小于Info<br>

\* 由于动态获取Logger，效率较低，建议在非频繁调用的情况下使用！！

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void debug(String format, Object... arguments) {

debug(innerGet(), format, arguments);

}

/\*\*

\* Debug等级日志，小于Info

\* @param log 日志对象

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void debug(Logger log, String format, Object... arguments) {

log.debug(format, arguments);

}

//------------------------ info

/\*\*

\* Info等级日志，小于Warn<br>

\* 由于动态获取Logger，效率较低，建议在非频繁调用的情况下使用！！

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void info(String format, Object... arguments) {

info(innerGet(), format, arguments);

}

/\*\*

\* Info等级日志，小于Warn

\* @param log 日志对象

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void info(Logger log, String format, Object... arguments) {

log.info(format, arguments);

}

//------------------------ warn

/\*\*

\* Warn等级日志，小于Error<br>

\* 由于动态获取Logger，效率较低，建议在非频繁调用的情况下使用！！

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void warn(String format, Object... arguments) {

warn(innerGet(), format, arguments);

}

/\*\*

\* Warn等级日志，小于Error

\* @param log 日志对象

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void warn(Logger log, String format, Object... arguments) {

log.warn(format, arguments);

}

/\*\*

\* Warn等级日志，小于Error<br>

\* 由于动态获取Logger，效率较低，建议在非频繁调用的情况下使用！！

\* @param e 需在日志中堆栈打印的异常

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void warn(Throwable e, String format, Object... arguments) {

warn(innerGet(), e, format(format, arguments));

}

/\*\*

\* Warn等级日志，小于Error

\* @param log 日志对象

\* @param e 需在日志中堆栈打印的异常

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void warn(Logger log, Throwable e, String format, Object... arguments) {

log.warn(format(format, arguments), e);

}

//------------------------ error

/\*\*

\* Error等级日志<br>

\* 由于动态获取Logger，效率较低，建议在非频繁调用的情况下使用！！

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void error(String format, Object... arguments) {

error(innerGet(), format, arguments);

}

/\*\*

\* Error等级日志<br>

\* @param log 日志对象

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void error(Logger log, String format, Object... arguments) {

log.error(format, arguments);

}

/\*\*

\* Error等级日志<br>

\* 由于动态获取Logger，效率较低，建议在非频繁调用的情况下使用！！

\* @param e 需在日志中堆栈打印的异常

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void error(Throwable e, String format, Object... arguments) {

error(innerGet(), e, format(format, arguments));

}

/\*\*

\* Error等级日志<br>

\* 由于动态获取Logger，效率较低，建议在非频繁调用的情况下使用！！

\* @param log 日志对象

\* @param e 需在日志中堆栈打印的异常

\* @param format 格式文本，{} 代表变量

\* @param arguments 变量对应的参数

\*/

public static void error(Logger log, Throwable e, String format, Object... arguments) {

log.error(format(format, arguments), e);

}

//----------------------------------------------------------- Logger method end

//----------------------------------------------------------- Private method start

/\*\*

\* 格式化文本

\* @param template 文本模板，被替换的部分用 {} 表示

\* @param values 参数值

\* @return 格式化后的文本

\*/

private static String format(String template, Object... values) {

return String.format(template.replace("{}", "%s"), values);

}

/\*\*

\* @return 获得日志，自动判定日志发出者

\*/

private static Logger innerGet() {

StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();

return LoggerFactory.getLogger(stackTrace[3].getClassName());

}

//----------------------------------------------------------- Private method end

}

Demo：

import org.slf4j.Logger;

import xxx.Log;

public class Test {

private static Logger log = Log.get();

public static void main(String[] args) {

//第一种使用方法（效率低）

Log.debug("我是一条debug消息");

//第二种使用方法

Log.debug(log, "我是一条debug消息 {} {}", "参数1", "参数2");

RuntimeException e = new RuntimeException("错误");

//第一种使用方法（效率低）

Log.error("我是一条error消息");

//第二种使用方法

Log.error(log, e, "<-异常对象放前面, 我是一条带参数的error消息 {} {}", "参数1", "参数2");

}

}

总结下来如果日志比较少，可以直接使用静态方法Log.xxx，如果日志量很大，那么首先要构建好Logger，使用：

```Java

private static Logger log = Log.get();

比之前简短了很多，而且随便复制。 调用Logger.xxx(log, "消息");

对于不能使用format的情况，我把Throwable放在前面了，这样就可以使用动态参数了Object... argument

好吧，就到这里，欢迎批评指正以及提供意见~~~

### 正则表达式工具类——ReUtil

#### 前言

在文本处理中，正则表达式几乎是全能的，但是Java的正则表达式有时候处理一些事情还是有些繁琐，所以我封装了部分常用功能。就比如说我要匹配一段文本中的某些部分，我们需要这样做：

Pattern pattern = Pattern.compile(regex, Pattern.DOTALL);

Matcher matcher = pattern.matcher(content);

if (matcher.find()) {

String result= matcher.group();

}

其中牵涉到多个对象，想用的时候真心记不住。好吧，既然功能如此常用，我就封装一下：

/\*\*

\* 获得匹配的字符串

\*

\* @param pattern 编译后的正则模式

\* @param content 被匹配的内容

\* @param groupIndex 匹配正则的分组序号

\* @return 匹配后得到的字符串，未匹配返回null

\*/

public static String get(Pattern pattern, String content, int groupIndex) {

Matcher matcher = pattern.matcher(content);

if (matcher.find()) {

return matcher.group(groupIndex);

}

return null;

}

/\*\*

\* 获得匹配的字符串

\*

\* @param regex 匹配的正则

\* @param content 被匹配的内容

\* @param groupIndex 匹配正则的分组序号

\* @return 匹配后得到的字符串，未匹配返回null

\*/

public static String get(String regex, String content, int groupIndex) {

Pattern pattern = Pattern.compile(regex, Pattern.DOTALL);

return get(pattern, content, groupIndex);

}

#### 其他方法介绍

其他方法我做了一个Demo类，方便大家参考，地址：

<https://github.com/looly/hutool/blob/master/src/test/java/com/xiaoleilu/hutool/demo/ReUtilDemo.java>

或者

<http://git.oschina.net/loolly/hutool/blob/master/src/test/java/com/xiaoleilu/hutool/demo/ReUtilDemo.java>

#### Demo代码

package com.xiaoleilu.hutool.demo;

import java.util.ArrayList;

import java.util.List;

import org.slf4j.Logger;

import com.xiaoleilu.hutool.Log;

import com.xiaoleilu.hutool.ReUtil;

/\*\*

\* 正则表达式工具类Demo

\* @author Looly

\*

\*/

public class ReUtilDemo {

private final static Logger log = Log.get();

public static void main(String[] args) {

String content = "ZZZaaabbbccc中文1234";

//get demo 正则查找匹配的第一个字符串

String resultGet = ReUtil.get("\\w{2}", content, 0);

log.debug("get: {}", resultGet);

log.debug("---------------------------------------------------------");

//抽取多个分组然后把它们拼接起来

String resultExtractMulti = ReUtil.extractMulti("(\\w)aa(\\w)", content, "$1-$2");

log.debug("extractMulti: {}", resultExtractMulti);

log.debug("---------------------------------------------------------");

//抽取多个分组然后把原文匹配到位置之前的内容都删除

String[] contents = new String[]{content};

String resultExtractMultiAndDelPre = ReUtil.extractMultiAndDelPre("(\\w)aa(\\w)", contents, "$1-$2");

log.debug("extractMultiAndDelPre: content: {}, extract: {}", contents[0], resultExtractMultiAndDelPre);

log.debug("---------------------------------------------------------");

//删除第一个匹配到的内容

String resultDelFirst = ReUtil.delFirst("(\\w)aa(\\w)", content);

log.debug("delFirst: {}", resultDelFirst);

log.debug("---------------------------------------------------------");

//删除第一个匹配到的内容以及之前的文本

String resultDelPre = ReUtil.delPre("(\\w)aa(\\w)", content);

log.debug("delPre: {}", resultDelPre);

log.debug("---------------------------------------------------------");

//查找所有匹配文本

List<String> resultFindAll = ReUtil.findAll("\\w{2}", content, 0, new ArrayList<String>());

log.debug("findAll: {}", resultFindAll);

log.debug("---------------------------------------------------------");

//找到匹配的第一个数字

Integer resultGetFirstNumber= ReUtil.getFirstNumber(content);

log.debug("getFirstNumber: {}", resultGetFirstNumber);

log.debug("---------------------------------------------------------");

//格式是否符合Ipv4格式

log.debug("isIpv4: {}", ReUtil.isIpv4("127.0.0.1"));

log.debug("---------------------------------------------------------");

//给定字符串是否匹配给定正则

log.debug("isMatch: {}", ReUtil.isMatch("\\w+[\u4E00-\u9FFF]+\\d+", content));

log.debug("---------------------------------------------------------");

//通过正则查找到字符串，然后把匹配到的字符串加入到replacementTemplate中，$1表示分组1的字符串

log.debug("replaceAll: {}", ReUtil.replaceAll(content, "(\\d+)", "->$1<-"));

log.debug("---------------------------------------------------------");

//转义给定字符串，为正则相关的特殊符号转义

log.debug("replaceAll: {}", ReUtil.escape("我有个$符号{}"));

log.debug("---------------------------------------------------------");

}

}