Hutool Wiki V4.1.0

前言

此文档已不再维护

请移步：https://hutool.cn/docs/(https://hutool.cn/docs/)

图片地址：https://static.oschina.net/uploads/space/2018/0907/212145_MMem_730640.png

• 项目主页：http://www.hutool.cn/(http://www.hutool.cn/)

• Hutool交流QQ群③：==555368316(http://shang.qq.com/wpa/qunwpa?idkey=35764b2247c46ffebe28e4541e5b2af8f5dee5efcf47ceec69d21e4521aa8c75) ==

• Hutool交流QQ群④：==718802356(http://shang.qq.com/wpa/qunwpa?idkey=309056e409a304a454c7ba250a10d38dd82b9b49cd0e1f180fedbde78b02ae0d) ==

简介

Hutool(https://github.com/looly/hutool)是Hu + tool的自造词，前者致敬我的“前任公司”，后者为工具之意，谐音“糊涂”，寓意追求“万事都作糊涂观，无所谓失，无所谓得”的境界。

Hutool是一个Java工具包，也只是一个工具包，它帮助我们简化每一行代码，减少每一个方法，让Java语言也可以“甜甜的”。Hutool最初是我项目中“util”包的一个整理，后来慢慢积累并加入更多非业务相关功能，并广泛学习其它开源项目精髓，经过自己整理修改，最终形成丰富的开源工具集。

功能

一个Java基础工具类，对文件、流、加密解密、转码、正则、线程、XML等JDK方法进行封装，组成各种Util工具类，同时提供以下组件：

• hutool-aop JDK动态代理封装，提供非IOC下的切面支持

• hutool-bloomFilter 布隆过滤，提供一些Hash算法的布隆过滤

• hutool-cache 缓存

• hutool-core 核心，包括Bean操作、日期、各种Util等

• hutool-cron 定时任务模块，提供类Crontab表达式的定时任务

- hutool-crypto 加密解密模块
- hutool-db JDBC封装后的数据操作，基于ActiveRecord思想
- hutool-dfa 基于DFA模型的多关键字查找
- hutool-extra 扩展模块，对第三方封装（模板引擎、邮件等）
- hutool-http 基于HttpUrlConnection的Http客户端封装
- hutool-log 自动识别日志实现的日志门面
- hutool-script 脚本执行封装，例如Javascript
- hutool-setting 功能更强大的Setting配置文件和Properties封装
- hutool-system 系统参数调用封装（JVM信息等）
- hutool-json JSON实现
- hutool-captcha 图片验证码实现
- hutool-poi 针对POI中Excel的封装

设计哲学

1. 方法优先于对象

在工具类中，往往以静态方法为主。方法集中在一个类中，配合IDE查找使用起来是十分便利的。于是Hutool将JDK中许多的类总结抽象为一个方法，这一原则使用最多的就是流的相关方法，这些方法很好的隐藏了XXXInputStream、XXXReader等的复杂性。

2. 自动识别优于用户定义

其实很多时候，有些参数、设置等是没有必要我们自己传入的，完全可以靠逻辑判断自动完成。一个方法很多时候明明只需要传3个参数，我们非要传4个，这多出的一个参数本身就是代码的一种冗余。

这一原则在Hutool的各个角落都有所体现，尤为明显的比如log模块。构建日志对象的时候，很明显类名可以动态获取，何必让使用者再传入呢？再比如在db模块的数据库配置中，数据库驱动命名完全可以根据连接字符串判断出来，何必要让用户传入？这些问题的在Hutool中都有非常好的封装，而这一原则也渐渐变成Hutool哲学的一部分。

3. 便捷性与灵活性并存

所谓便捷性，就是我们在调用一个方法的时候参数要尽量少，只传必要参数即可，非必要参数使用默认值即可（想想一个方法一堆参数的时候，调用者晕头转向不知所云）。

所谓灵活性正好与便捷性相反，要让一个方法的参数尽量多，为用户灵活的操作方法提供最大可能性。

这两个原则看似矛盾，其实只是针对不同场景设定的而已，缺一不可。便捷性强调拿来即用，为快速开发提供可能；灵活性强调最大限度调优，为性能调优和扩展提供便利。

这一原则在针对编码问题上体现尤为突出，我们的大部分方法都是默认“UTF-8”编码的，这也是我们推荐的编码方式，推荐大部分项目使用的编码。但是一旦有遗留项目使用了类似“GBK”等编码，没关系，我们提供在相关方法中提供Charset对象参数，可以自定义编码。这样使用这一原则

就兼顾了各种项目的情况。

4. 适配与兼容

在Hutool中，适配器模式运用特别广泛，log模块适配主流各大框架，db模块适配主流各种连接池和关系数据库。这种适配一是提高灵活性，二是可以很好的兼容各大框架，让Hutool可以在各种复杂项目环境中生存的很好。

适配兼容产生的另一个原则是：**你有我配，你无我有。**说白了就是：如果你项目中有这个框架，我可以完美适配，如果你没有引入任何框架，Hutool自身实现了一些逻辑可以很好的工作。

5. 可选依赖原则

在Java项目中依赖常常是个头疼的问题，不同的框架强依赖另一些框架或包，虽然Maven可以很好的处理冲突问题，但是项目底下满满的依赖jar包，是不是无形中拖慢了项目，也增加了复杂性和不确定性？而很多时候，我们是不是只是为了用一个小小的方法，就要引入一个第三方包，谁喜欢这样臃肿的项目？

Hutool中也会有一些依赖，但是全部都是**optional**的，在使用中不会关联依赖，而这些依赖只有在使用者使用到时才会调用，这时可能会报ClassNotFoundException，不用担心，我们自己引入即可。为什么要这样做呢？以VelocityUtil这个工具类为例，使用Velocity的人占比极少，我们不能为了这些用户而强引入Velocity包，而使用这个工具类的人应该明白，我们应该自己引入这个包。

而更多时候，我们需要用到某个方法时，我的做法是将方法拷贝到项目中（Hutool中的方法正在不断积累），类似于Apache Commons中的方法，Hutool中基本都有取代方法，完全不必要引入。

可选依赖原则让我们的项目更加精简，问题也更容易排查。

6. 无侵入原则

Hutool始终是一个工具类而不是框架，这意味着它对项目的侵入几乎为零，每个方法都是可被代替的，甚至整个Hutool也是可被替换的。这种无侵入性，让使用者可以更加放心的在项目中引入，也保证了与其它框架完美的兼容。

添砖加瓦

遵照的原则

我欢迎任何人为Hutool添砖加瓦，贡献代码，不过作者是一个强迫症患者，为了照顾病人，需要提交的pr（pull request）符合一些规范，规范如下：

1. 注释完备，尤其每个新增的方法应按照Java文档规范标明方法说明、参数说明、返回值说明等信息，如果愿意，也可以加上你的大名。

2. Hutool的缩进按照Eclipse（不要跟我说IDEA多好用，作者非常懒，学不会）默认（tab）缩进，所以请遵守（不要和我争执空格与tab的问题，这是一个病人的习惯）。

3. 新加的方法不要使用第三方库的方法，Hutool遵循无依赖原则（除非在extra模块中加方法工具）。

4. Hutool在4.x版本后使用了新的分支：v4-master是主分支，表示已经发布中央库的版本，这个分支不允许pr，也不允许修改。v4-dev分支是开发分支，Hutool的下个版本或者SNAPSHOT版本在这个分支上开发，你可以pr到这个分支。

贡献代码的方法

1. 在Gitee或者Github上fork项目到自己的repo
2. 把fork过去的项目也就是你的项目clone到你的本地
3. 修改代码（记得一定要修改v4-dev分支）
4. commit后push到自己的库（v4-dev分支）
5. 登录Gitee或Github在你首页可以看到一个 pull request 按钮，点击它，填写一些说明信息，然后提交即可。
6. 等待作者合并

版本变更

Release版本变更说明

https://gitee.com/loolly/hutool/blob/v4-master/CHANGELOG.md(https://gitee.com/loolly/hutool/blob/v4-master/CHANGELOG.md)

Snapshot版本变更说明

https://gitee.com/loolly/hutool/blob/v4-dev/CHANGELOG.md(https://gitee.com/loolly/hutool/blob/v4-dev/CHANGELOG.md)

安装

> 注意
>
> Hutool只支持JDK7+，对应Android平台没有测试，部分方法并不支持。

子模块

每个子模块可以被单独引入使用，修改左边名字为artifactId即可。如果想引入所有模块，可以引入hutool-all即可。

- hutool-core 核心，包括Bean操作、日期、各种Util等
- hutool-aop 动态代理封装，提供非IOC下的切面支持
- hutool-bloomFilter 布隆过滤，提供一些Hash算法的布隆过滤
- hutool-cache 缓存

- hutool-captcha 图片验证码实现
- hutool-cron 定时任务模块，提供类Crontab表达式的定时任务
- hutool-crypto 加密解密模块
- hutool-db JDBC封装后的数据操作，基于ActiveRecord思想
- hutool-dfa 基于DFA模型的多关键字查找（敏感词匹配和替换）
- hutool-extra 扩展模块，对第三方封装（模板引擎、邮件、FTP、二维码等）
- hutool-http 基于HttpUrlConnection的Http客户端封装
- hutool-log 自动识别日志实现的日志门面
- hutool-script 脚本执行封装，例如Javascript
- hutool-setting 功能更强大的Setting配置文件和Properties封装
- hutool-system 系统参数调用封装（JVM信息等）
- hutool-json JSON实现
- hutool-poi 针对POI中Excel的封装

通过Maven引入项目

RELEASE版本引用

在项目的pom.xml的dependencies中加入以下内容:

```
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>${hutool.version}</version>
</dependency>
```

SNAPSHOT版本引用

1. 添加SNAPSHOT支持：

```
<parent>
    <groupId>org.sonatype.oss</groupId>
    <artifactId>oss-parent</artifactId>
    <version>9</version>
</parent>
```

1. 引用SNAPSHOT版本的库

```
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
```

```
    <version>${hutool.version}-SNAPSHOT</version>
</dependency>
```

通过Gradle引入项目

```
compile 'cn.hutool:hutool-all:${hutool.version}'
```

> 工具包的**版本**可以通过 http://search.maven.org/(http://search.maven.org/) 搜索hutool-all找到项目。

非Maven项目

直接下载jar包

可以从http://search.maven.org/(http://search.maven.org/) 搜索hutool-all找到项目，点击对应版本，下面是相应的Jar包，导入即可使用。

一键直达：http://repo2.maven.org/maven2/cn/hutool/hutool-all/4.1.19/(http://repo2.maven.org/maven2/cn/hutool/hutool-all/4.1.19/)

自行编译安装

访问Hutool的码云主页：https://gitee.com/loolly/hutool(https://gitee.com/loolly/hutool) 下载整个项目源码（v4-master或v4-dev分支都可）然后进入Hutool项目目录执行：

```
bin/hutool.sh install
```

友情开源项目

feilong-core

项目地址：https://github.com/venusdrogon/feilong-core

作者：飞天奔月

feilong-core最在开源中国中发现的，与Hutool实现相同的目的，而采用不同的方式。后来与作者@飞天奔月交流甚多。Hutool在feilong-core中也吸取了很多非常好的思想。

图片地址：https://static.oschina.net/uploads/img/201707/11151000_Glow.png

---

t-io

项目地址：http://git.oschina.net/tywo45/t-io

作者：talent-tan

t-io也就是原来的talent-aio，是一个基于AIO的即时通讯框架，作者@talent-tan也是一个非常热心肠的人，素未谋面却聊的很投缘。也感谢作者把Hutool-core引入t-io作为依赖。

图片地址：https://static.oschina.net/uploads/img/201704/28172303_IYXt.jpg

---

ActFramework

项目地址：http://git.oschina.net/actframework/actframework

作者：greenlaw110

ActFramework是一个抛弃了Servlet规范的MVC框架，这一点让我眼前一亮，而且这个框架背后还有一些作者写的工具类支撑，这些工具类别出心裁（比如用单字母做为类名，或者用$做为类名），让我大开眼界。同时作者也是非常热心的人，我们相互提了一些bug，同时Hutool的一些工具类也参考了作者框架中的一些方法。在此表示感谢。

Voovan

项目地址：http://git.oschina.net/helyho/Voovan

作者：愚民日记

Voovan更像是t-io的竞争者（其实两位作者好的不得了），同时又集成了好多工具类，因此很早以前就被我“盯上”了，后来逐渐通过Hutool群认识作者，相见恨晚。Hutool中也有很多工具方法参考了Voovan中的思想，感谢作者。

Hutool相关博客（软文）

Hutool的博客和软文

• 安利一波那个叫做 Hutool 的通用工具类库(http://www.cnblogs.com/java-class/p/7742481.html)

• Java工具方法hutool 使用备注(http://blog.csdn.net/earbao/article/details/46832853)

• Hutool是一个JAVA工具包(http://www.jfinal.com/share/252)

• hutool java工具架包功能介绍(https://blog.csdn.net/lx1309244704/article/details/76459718)

• 【Hutool】Hutool工具类之Http工具——HttpUtil(https://www.cnblogs.com/jiangbei/p/7667858.html)

• Hutool-Java基础工具包(https://mp.weixin.qq.com/s?src=11&timestamp=1521945641&ver=775&signature=TCm61hzYfoFK8TjtgS3RkCtf7h1AXpC1ONMoImcMqpiF5o5oJOa5e51pM8H4x36wJWWKfgt3HmYEagdGXFKWdHhfh5WYbveXkQed5SkYvYCWKIWfg6we**QexadhzPqR&new=1)

• Hutool-贼好用的Java工具类库(https://ryanc.cc/archives/hutool-java-tools-lib)

使用Hutool的开源项目

Fast-boot(https://gitee.com/jiangzeyin/common-parent)

Halo(https://github.com/ruibaby/halo)

捐赠使用公开

介绍

在过去的几年中，Hutool的发展离不开用户的支持，每一个issue，每一句吐槽都是Hutool进步的动力，不只这些，在Gitee中开启捐赠后，有好多的支持者为Hutool捐赠，支持Hutool的发展。为了不辜负广大用户的期望，特意在文档中开设此页面公开捐赠使用情况，如果对此有异议也可以加Hutool群控诉作者哦。

当然这里的捐赠使用情况并不完全准确，不排除我会用捐赠的钱买雪糕和辣条哦。

捐赠地址

https://gitee.com/loolly/hutool(https://gitee.com/loolly/hutool)

2017年

用途	金额
hutool.cn域名费用（3年）	￥105

2018年

用途	金额
Hutool群新春福利红包	￥100
Hutool群VIP会员费用	￥228
作者偷吃辣条一包	￥2.5

核心(Hutool-core)

克隆

支持泛型的克隆接口和克隆类

我们解决什么问题

我们知道，JDK中的Cloneable接口只是一个空接口，并没有定义成员，它存在的意义仅仅是指明一个类的实例化对象支持位复制（就是对象克隆），如果不实现这个类，调用对象的clone()方法就会抛出CloneNotSupportedException异常。而且，因为clone()方法在Object对象中，返回值也是Object对象，因此克隆后我们需要自己强转下类型。

泛型克隆接口

因此，**com.xiaoleilu.hutool.clone.Cloneable<T>**接口应运而生。此接口定义了一个返回泛型的成员方法，这样，实现此接口后会提示必须实现一个public的clone方法，调用父类clone方法即可：

```
/**
 * 猫猫类，使用实现Cloneable方式
 * @author Looly
 *
 */
private static class Cat implements Cloneable<Cat>{
private String name = "miaomiao";
private int age = 2;

@Override
public Cat clone() {
try {
return (Cat) super.clone();
} catch (CloneNotSupportedException e) {
throw new CloneRuntimeException(e);
}
}
}
```

泛型克隆类

但是实现此接口依旧有不方便之处，就是必须自己实现一个public类型的clone()方法，还要调用父类（Object）的clone方法并处理异常。于是**com.xiaoleilu.hutool.clone.CloneSupport<T>**类产生，这个类帮我们实现了上面的clone方法，因此只要继承此类，不用写任何代码即可使用clone()方法：

```
/**
```

```
 * 狗狗类，用于继承CloneSupport类
 * @author Looly
 *
 */
private static class Dog extends CloneSupport<Dog>{
private String name = "wangwang";
private int age = 3;
}
```

当然，使用**CloneSupport**的前提是你没有继承任何的类，谁让Java不支持多重继承呢（你依旧可以让父类实继承这个类，如果可以的话）。如果没办法继承类，那实现**com.xiaoleilu.hutool.clone.Cloneable<T>**也是不错的主意，因此**hutool**提供了这两种方式，任选其一，在便捷和灵活上都提供了支持。

深克隆

我们知道实现Cloneable接口后克隆的对象是浅克隆，要想实现深克隆，请使用：

```
ObjectUtil.cloneByStream(obj)
```

前提是对象必须实现Serializable接口。

ObjectUtil同样提供一些静态方法：**clone(obj)**、**cloneIfPossible(obj)**用于简化克隆调用，详细的说明请查看核心类的相关文档。

类型转换

类型转换工具类-Convert

痛点

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。

Convert类

Convert类可以说是一个工具方法类，里面封装了针对Java常见类型的转换，用于简化类型转换。**Convert**类中大部分方法为toXXX，参数为Object，可以实现将任意可能的类型转换为指定类型。同时支持第二个参数**defaultValue**用于在转换失败时返回一个默认值。

Java常见类型转换

1、转换为字符串：

```
int a = 1;
//aStr为"1"
String aStr = Convert.toStr(a);

long[] b = {1,2,3,4,5};
//bStr为："[1, 2, 3, 4, 5]"
String bStr = Convert.toStr(b);
```

2、转换为指定类型数组：

```
String[] b = { "1", "2", "3", "4" };
//结果为Integer数组
Integer[] intArray = Convert.toIntArray(b);

long[] c = {1,2,3,4,5};
//结果为Integer数组
Integer[] intArray2 = Convert.toIntArray(c);
```

3、转换为日期对象：

```
String a = "2017-05-06";
Date value = Convert.toDate(a);
```

4、转换为集合

```
Object[] a = {"a", "你", "好", "", 1};
List<?> list = Convert.convert(List.class, a);
//从4.1.11开始可以这么用
List<?> list = Convert.toList(a);
```

其它类型转换

通过Convert.convert(Class<T>, Object)方法可以将任意类型转换为指定类型，Hutool中预定义了许多类型转换，例如转换为URI、URL、Calendar等等，这些类型的转换都依托于ConverterRegistry类。通过这个类和Converter接口，我们可以自定义一些类型转换。详细的使用请参阅“自定义类型转换”一节。

半角和全角转换

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。

半角转全角：

```
String a = "123456789";

//结果为："１２３４５６７８９"
```

```
String sbc = Convert.toSBC(a);
```

全角转半角：

```
String a = "１２３４５６７８９";

//结果为"123456789"
String dbc = Convert.toDBC(a);
```

16进制（Hex）

在很多加密解密，以及中文字符串传输（比如表单提交）的时候，会用到16进制转换，就是Hex转换，为此Hutool中专门封装了**HexUtil**工具类，考虑到16进制转换也是转换的一部分，因此将其方法也放在Convert类中，便于理解和查找，使用同样非常简单：

转为16进制（Hex）字符串

```
String a = "我是一个小小的可爱的字符串";

//结果
：
"e68891e698afe4b880e4b8aae5b08fe5b08fe79a84e58fafe788b1e79a84e5ad97e7aca6e4b8b2"
String hex = Convert.toHex(a, CharsetUtil.CHARSET_UTF_8);
```

将16进制（Hex）字符串转为普通字符串:

```
String hex = "e68891e698afe4b880e4b8aae5b08fe5b08fe79a84e58fafe788b1e79a84e5ad97e7aca6e4b8b2";

//结果为："我是一个小小的可爱的字符串"
String raw = Convert.hexStrToStr(hex, CharsetUtil.CHARSET_UTF_8);

//注意：在4.1.11之后hexStrToStr将改名为hexToStr
String raw = Convert.hexToStr(hex, CharsetUtil.CHARSET_UTF_8);
```

> 因为字符串牵涉到编码问题，因此必须传入编码对象，此处使用UTF-8编码。
>
> **toHex**方法同样支持传入byte[]，同样也可以使用**hexToBytes**方法将16进制转为byte[]

Unicode和字符串转换

与16进制类似，Convert类同样可以在字符串和Unicode之间轻松转换：

```
String a = "我是一个小小的可爱的字符串";
```

```
//结果为
：\"\\u6211\\u662f\\u4e00\\u4e2a\\u5c0f\\u5c0f\\u7684\\u53ef\\u7231\\u7684\\u5b57\\u7b26\\u4e32"
String unicode = Convert.strToUnicode(a);

//结果为："我是一个小小的可爱的字符串"
String raw = Convert.unicodeToStr(unicode);
```

很熟悉吧？如果你在properties文件中写过中文，你会明白这个方法的重要性。

编码转换

在接收表单的时候，我们常常被中文乱码所困扰，其实大多数原因是使用了不正确的编码方式解码了数据。于是Convert.convertCharset方法便派上用场了，它可以把乱码转为正确的编码方式：

```
String a = "我不是乱码";
//转换后result为乱码
String result = Convert.convertCharset(a, CharsetUtil.UTF_8, CharsetUtil.ISO_8859_1);
String raw = Convert.convertCharset(result, CharsetUtil.ISO_8859_1, "UTF-8");
Assert.assertEquals(raw, a);
```

> 注意
>
> 经过测试，UTF-8编码后用GBK解码再用GBK编码后用UTF-8解码会存在某些中文转换失败的问题。

时间单位转换

Convert.convertTime方法主要用于转换时长单位，比如一个很大的毫秒，我想获得这个毫秒数对应多少分：

```
long a = 4535345;

//结果为：75
long minutes = Convert.convertTime(a, TimeUnit.MILLISECONDS, TimeUnit.MINUTES);
```

金额大小写转换

面对财务类需求，Convert.digitToChinese将金钱数转换为大写形式：

```
double a = 67556.32;

//结果为："陆万柒仟伍佰伍拾陆元叁角贰分"
String digitUppercase = Convert.digitToChinese(a);
```

“

注意

转换为大写只能精确到分（小数点儿后两位），之后的数字会被忽略。

原始类和包装类转换

有的时候，我们需要将包装类和原始类相互转换（比如Integer.classs 和 int.class），这时候我们可以：

```
//去包装
Class<?> wrapClass = Integer.class;

//结果为：int.class
Class<?> unWraped = Convert.unWrap(wrapClass);

//包装
Class<?> primitiveClass = long.class;

//结果为：Long.class
Class<?> wraped = Convert.wrap(primitiveClass);
```

自定义类型转换-ConverterRegistry

由来

Hutool中类型转换最早只是一个工具类，叫做“Conver”，对于每一种类型转换都是用一个静态方法表示，但是这种方式有一个潜在问题，那就是扩展性不足，这导致Hutool只能满足部分类型转换的需求。

解决

为了解决这些问题，我对Hutool中这个类做了扩展。思想如下：

- Converter 类型转换接口，通过实现这个接口，重写convert方法，以实现不同类型的对象转换
- ConverterRegistry 类型转换登记中心。将各种类型Convert对象放入登记中心，通过convert方法查找目标类型对应的转换器，将被转换对象转换之。在此类中，存放着**默认转换器**和**自定义转换器**，默认转换器是Hutool中预定义的一些转换器，自定义转换器存放用户自定的转换器。

通过这种方式，实现类灵活的类型转换。使用方式如下：

```
int a = 3423;
ConverterRegistry converterRegistry = ConverterRegistry.getInstance();
String result = converterRegistry.convert(String.class, a);
Assert.assertEquals("3423", result);
```

自定义转换

Hutool的默认转换有时候并不能满足我们自定义对象的一些需求，这时我们可以使用ConverterRegistry.getInstance().putCustom()方法自定义类型转换。

1. 自定义转换器

```
public static class CustomConverter implements Converter<String>{
@Override
public String convert(Object value, String defaultValue) throws
IllegalArgumentException {
return "Custom: " + value.toString();
}
}
```

1. 注册转换器

```
ConverterRegistry converterRegistry = ConverterRegistry.getInstance();
//此处做为示例自定义String转换，因为Hutool中已经提供String转换，请尽量不要替换
//替换可能引发关联转换异常（例如覆盖String转换会影响全局）
converterRegistry.putCustom(String.class, CustomConverter.class);
```

1. 执行转换

```
int a = 454553;
String result = converterRegistry.convert(String.class, a);
Assert.assertEquals("Custom: 454553", result);
```

> 注意：
>
> convert(Class<T> type, Object value, T defaultValue, boolean isCustomFirst)方法的最后一个参数可以选择转换时优先使用自定义转换器还是默认转换器。convert(Class<T> type, Object value, T defaultValue)和convert(Class<T> type, Object value)两个重载方法都是使用自定义转换器优先的模式。

ConverterRegistry单例和对象模式

ConverterRegistry提供一个静态方法getInstance()返回全局单例对象，这也是推荐的使用方式，当然如果想在某个限定范围内自定义转换，可以实例化ConverterRegistry对象。

日期时间

概述

日期时间包是Hutool的核心包之一，提供针对JDK中Date和Calendar对象的封装，封装对象如下：

日期时间工具

• DateUtil 针对日期时间操作提供一系列静态方法

• DateTime 提供类似于Joda-Time中日期时间对象的封装，继承自Date类，并提供更加丰富的对象方法。

• FastDateFormat 提供线程安全的针对Date对象的格式化和日期字符串解析支持。此对象在实际使用中并不需要感知，相关操作已经封装在DateUtil和DateTime的相关方法中。

• DateBetween 计算两个时间间隔的类，除了通过构造新对象使用外，相关操作也已封装在DateUtil和DateTime的相关方法中。

• TimeInterval 一个简单的计时器类，常用于计算某段代码的执行时间，提供包括毫秒、秒、分、时、天、周等各种单位的花费时长计算，对象的静态构造已封装在DateUtil中。

• DatePattern 提供常用的日期格式化模式，包括String类型和FastDateFormat两种类型。

日期枚举

考虑到Calendar类中表示时间的字段（field）都是使用int表示，在使用中非常不便，因此针对这些int字段，封装了与之对应的Enum枚举类，这些枚举类在DateUtil和DateTime相关方法中做为参数使用，可以更大限度的缩小参数限定范围。

这些定义的枚举值可以通过getValue()方法获得其与Calendar类对应的int值，通过of(int)方法从Calendar中int值转为枚举对象。

与Calendar对应的这些枚举包括：

• Month 表示月份，与Calendar中的int值一一对应。

• Week 表示周，与Calendar中的int值一一对应

另外，Hutool还定义了**季度**枚举。Season.SPRING为第一季度，表示1~3月。季度的概念并不等同于季节，因为季节与月份并不对应，季度常用于统计概念。

时间枚举

时间枚举DateUnit主要表示某个时间单位对应的毫秒数，常用于计算时间差。

例如：DateUnit.MINUTE表示分，也表示一分钟的毫米数，可以通过调用其getMillis()方法获得其毫秒数。

日期时间工具-DateUtil

由来

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，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），提供了众多便捷的方法对日期对象操作，关于DateTime会在相关章节介绍。

方法

转换

Date、long、Calendar之间的相互转换

```
//当前时间
Date date = DateUtil.date();
//当前时间
Date date2 = DateUtil.date(Calendar.getInstance());
//当前时间
Date date3 = DateUtil.date(System.currentTimeMillis());
//当前时间字符串，格式：yyyy-MM-dd HH:mm:ss
String now = DateUtil.now();
//当前日期字符串，格式：yyyy-MM-dd
String today= DateUtil.today();
```

字符串转日期

DateUtil.parse方法会自动识别一些常用格式，包括：

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

```
String dateStr = "2017-03-01";
Date date = DateUtil.parse(dateStr);
```

我们也可以使用自定义日期格式转化：

```
String dateStr = "2017-03-01";
Date date = DateUtil.parse(dateStr, "yyyy-MM-dd");
```

格式化日期输出

```
String dateStr = "2017-03-01";
Date date = DateUtil.parse(dateStr);

//结果 2017/03/01
String format = DateUtil.format(date, "yyyy/MM/dd");
```

```
//常用格式的格式化，结果：2017-03-01
String formatDate = DateUtil.formatDate(date);

//结果：2017-03-01 00:00:00
String formatDateTime = DateUtil.formatDateTime(date);

//结果：00:00:00
String formatTime = DateUtil.formatTime(date);
```

获取Date对象的某个部分

```
Date date = DateUtil.date();
//获得年的部分
DateUtil.year(date);
//获得月份，从0开始计数
DateUtil.month(date);
//获得月份枚举
DateUtil.monthEnum(date);
//.....
```

开始和结束时间

有的时候我们需要获得每天的开始时间、结束时间，每月的开始和结束时间等等，DateUtil也提供了相关方法：

```
String dateStr = "2017-03-01 22:33:23";
Date date = DateUtil.parse(dateStr);

//一天的开始，结果：2017-03-01 00:00:00
Date beginOfDay = DateUtil.beginOfDay(date);

//一天的结束，结果：2017-03-01 23:59:59
Date endOfDay = DateUtil.endOfDay(date);
```

日期时间偏移

日期或时间的偏移指针对某个日期增加或减少分、小时、天等等，达到日期变更的目的。Hutool也针对其做了大量封装

```
String dateStr = "2017-03-01 22:33:23";
Date date = DateUtil.parse(dateStr);
```

```
//结果：2017-03-03 22:33:23
Date newDate = DateUtil.offset(date, DateField.DAY_OF_MONTH, 2);

//常用偏移，结果：2017-03-04 22:33:23
DateTime newDate2 = DateUtil.offsetDay(date, 3);

//常用偏移，结果：2017-03-01 19:33:23
DateTime newDate3 = DateUtil.offsetHour(date, -3);
```

针对当前时间，提供了简化的偏移方法（例如昨天、上周、上个月等）：

```
//昨天
DateUtil.yesterday()
//明天
DateUtil.tomorrow()
//上周
DateUtil.lastWeek()
//下周
DateUtil.nextWeek()
//上个月
DateUtil.lastMonth()
//下个月
DateUtil.nextMonth()
```

日期时间差

有时候我们需要计算两个日期之间的时间差（相差天数、相差小时数等等），Hutool将此类方法封装为between方法：

```
String dateStr1 = "2017-03-01 22:33:23";
Date date1 = DateUtil.parse(dateStr1);

String dateStr2 = "2017-04-01 23:33:23";
Date date2 = DateUtil.parse(dateStr2);

//相差一个月，31天
long betweenDay = DateUtil.between(date1, date2, DateUnit.DAY);
```

格式化时间差

有时候我们希望看到易读的时间差，比如XX天XX小时XX分XX秒，此时使用DateUtil.formatBetween方法：

```
//Level.MINUTE表示精确到分
String formatBetween = DateUtil.formatBetween(between, Level.MINUTE);
//输出：31天1小时
Console.log(formatBetween);
```

计时器

计时器用于计算某段代码或过程花费的时间

```
TimeInterval timer = DateUtil.timer();

//----------------------------------
//-------这是执行过程
//----------------------------------

timer.interval();//花费毫秒数
timer.intervalRestart();//返回花费时间，并重置开始时间
timer.intervalMinute();//花费分钟数
```

其它

```
//年龄
DateUtil.ageOfNow("1990-01-30");

//是否闰年
DateUtil.isLeapYear(2017);
```

日期时间对象-DateTime

由来

考虑工具类的局限性，在某些情况下使用并不简便，于是DateTime类诞生。DateTime对象充分吸取Joda-Time库的优点，并提供更多的便捷方法，这样我们在开发时不必再单独导入Joda-Time库便可以享受简单快速的日期时间处理过程。

说明

DateTime类继承于java.util.Date类，为Date类扩展了众多简便方法，这些方法多是DateUtil静态方法的对象表现形式，使用DateTime对象可以完全替代开发中Date对象的使用。

使用

新建对象

DateTime对象包含众多的构造方法，构造方法支持的参数有：

- Date
- Calendar
- String(日期字符串，第二个参数是日期格式)
- long 毫秒数

构建对象有两种方式：DateTime.of()和new DateTime()：

```
Date date = new Date();

//new方式创建
DateTime time = new DateTime(date);
Console.log(time);

//of方式创建
DateTime now = DateTime.now();
DateTime dt = DateTime.of(date);
```

使用对象

DateTime的成员方法与DateUtil中的静态方法所对应，因为是成员方法，因此可以使用更少的参数操作日期时间。

示例：获取日期成员（年、月、日等）

```
DateTime dateTime = new DateTime("2017-01-05 12:34:23",
DatePattern.NORM_DATETIME_FORMAT);

//年，结果：2017
int year = dateTime.year();

//季度（非季节），结果：Season.SPRING
Season season = dateTime.seasonEnum();

//月份，结果：Month.JANUARY
Month month = dateTime.monthEnum();

//日，结果：5
int day = dateTime.dayOfMonth();
```

更多成员方法请参阅API文档。

对象的可变性

DateTime对象默认是可变对象（调用offset、setField、setTime方法默认变更自身），但是这种可变性有时候会引起很多问题（例如多个地方共用DateTime对象）。我们可以调用setMutable(false)方法使其变为不可变对象。在不可变模式下，offset、setField方法返回一个新对象，setTime方法抛出异常。

```
DateTime dateTime = new DateTime("2017-01-05 12:34:23",
DatePattern.NORM_DATETIME_FORMAT);

//默认情况下DateTime为可变对象，此时offsite == dateTime
DateTime offsite = dateTime.offsite(DateField.YEAR, 0);

//设置为不可变对象后变动将返回新对象，此时offsite != dateTime
dateTime.setMutable(false);
offsite = dateTime.offsite(DateField.YEAR, 0);
```

格式化为字符串

调用toString()方法即可返回格式为yyyy-MM-dd HH:mm:ss的字符串，调用toString(String format)可以返回指定格式的字符串。

```
DateTime dateTime = new DateTime("2017-01-05 12:34:23",
DatePattern.NORM_DATETIME_FORMAT);
//结果：2017-01-05 12:34:23
String dateStr = dateTime.toString();

//结果：2017/01/05
```

IO

概述

由来

IO的操作包括**读**和**写**，应用场景包括网络操作和文件操作。IO操作在Java中是一个较为复杂的过程，我们在面对不同的场景时，要选择不同的InputStream和OutputStream实现来完成这些操作。而如果想读写字节流，还需要Reader和Writer的各种实现类。这些繁杂的实现类，一方面给我我们提供了更多的灵活性，另一方面也增加了复杂性。

封装

io包的封装主要针对流、文件的读写封装，主要以工具类为主，提供常用功能的封装，这包括：

- IoUtil 流操作工具类

- FileUtil 文件读写和操作的工具类。
- FileTypeUtil 文件类型判断工具类
- WatchMonitor 目录、文件监听，封装了JDK1.7中的WatchService
- ClassPathResource针对ClassPath中资源的访问封装
- FileReader 封装文件读取
- FileWriter 封装文件写入

流扩展

除了针对JDK的读写封装外，还针对特定环境和文件扩展了流实现。

包括：

- BOMInputStream针对含有BOM头的流读取
- FastByteArrayOutputStream 基于快速缓冲FastByteBuffer的OutputStream，随着数据的增长自动扩充缓冲区（from blade）
- FastByteBuffer 快速缓冲，将数据存放在缓冲集中，取代以往的单一数组（from blade）

IO工具类-IoUtil

由来

IO工具类的存在主要针对InputStream、OutputStream、Reader、Writer封装简化，并对NIO相关操作做封装简化。总体来说，Hutool对IO的封装，主要是工具层面，我们努力做到在便捷、性能和灵活之间找到最好的平衡点。

方法

拷贝

流的读写可以总结为从输入流读取，从输出流写出，这个过程我们定义为**拷贝**。这个是一个基本过程，也是文件、流操作的基础。

以文件流拷贝为例：

```
BufferedInputStream in = FileUtil.getInputStream("d:/test.txt");
BufferedOutputStream out = FileUtil.getOutputStream("d:/test2.txt");
long copySize = IoUtil.copy(in, out, IoUtil.DEFAULT_BUFFER_SIZE);
```

copy方法同样针对Reader、Writer、Channel等对象有一些重载方法，并提供可选的缓存大小。默认的，缓存大小为1024个字节，如果拷贝大文件或流数据较大，可以适当调整这个参数。

针对NIO，提供了copyByNIO方法，以便和BIO有所区别。我查阅过一些资料，使用NIO对文件流的操作有一定的提升，我并没有做具体实验。相关测试请参阅博客：http://www.cnblogs.com/gaopeng527/p/4896783.html(http://www.cnblogs.com/gaopeng527/p/4896783.html)

Stream转Reader、Writer

• IoUtil.getReader：将InputStream转为BufferedReader用于读取字符流，它是部分readXXX方法的基础。

• IoUtil.getWriter：将OutputStream转为OutputStreamWriter用于写入字符流，它是部分writeXXX的基础。

本质上这两个方法只是简单new一个新的Reader或者Writer对象，但是封装为工具方法配合IDE的自动提示可以大大减少查阅次数（例如你对BufferedReader、OutputStreamWriter不熟悉，是不需要搜索一下相关类？）

读取流中的内容

读取流中的内容总结下来，可以分为read方法和readXXX方法。

1. read方法有诸多的重载方法，根据参数不同，可以读取不同对象中的内容，这包括：

• InputStream

• Reader

• FileChannel

这三个重载大部分返回String字符串，为字符流读取提供极大便利。

1. readXXX方法主要针对返回值做一些处理，例如：

• readBytes 返回byte数组（读取图片等）

• readHex 读取16进制字符串

• readObj 读取序列化对象（反序列化）

• readLines 按行读取

1. toStream方法则是将某些对象转换为流对象，便于在某些情况下操作：

• String 转换为ByteArrayInputStream

• File 转换为FileInputStream

写入到流

• IoUtil.write方法有两个重载方法，一个直接调用OutputStream.write方法，另一个用于将对象转换为字符串（调用toString方法），然后写入到流中。

• IoUtil.writeObjects 用于将可序列化对象序列化后写入到流中。

write方法并没有提供writeXXX，需要自己转换为String或byte[]。

关闭

对于IO操作来说，使用频率最高（也是最容易被遗忘）的就是close操作，好在Java规范使用了优雅的Closeable接口，这样我们只需简单封装调用此接口的方法即可。

关闭操作会面临两个问题：

1. 被关闭对象为空

2. 对象关闭失败（或对象已关闭）

IoUtil.close方法很好的解决了这两个问题。

在JDK1.7中，提供了AutoCloseable接口，在IoUtil中同样提供相应的重载方法，在使用中并不能感觉到有哪些不同。

文件工具类-FileUtil

简介

在IO操作中，文件的操作相对来说是比较复杂的，但也是使用频率最高的部分，我们几乎所有的项目中几乎都躺着一个叫做FileUtil或者FileUtils的工具类，我想Hutool应该将这个工具类纳入其中，解决用来解决大部分的文件操作问题。

总体来说，FileUtil类包含以下几类操作工具：

1. 文件操作：包括文件目录的新建、删除、复制、移动、改名等
2. 文件判断：判断文件或目录是否非空，是否为目录，是否为文件等等。
3. 绝对路径：针对ClassPath中的文件转换为绝对路径文件。
4. 文件名：主文件名，扩展名的获取
5. 读操作：包括类似IoUtil中的getReader、readXXX操作
6. 写操作：包括getWriter和writeXXX操作

在FileUtil中，我努力将方法名与Linux相一致，例如创建文件的方法并不是createFile，而是touch，这种统一对于熟悉Linux的人来说，大大提高了上手速度。当然，如果你不熟悉Linux，那FileUtil工具类的使用则是在帮助你学习Linux命令。这些类Linux命令的方法包括：

- ls 列出目录和文件
- touch 创建文件，如果父目录不存在也自动创建
- mkdir 创建目录，会递归创建每层目录
- del 删除文件或目录（递归删除，不判断是否为空），这个方法相当于Linux的delete命令
- copy 拷贝文件或目录

这些方法提供了人性化的操作，例如touch方法，在创建文件的情况下会自动创建上层目录（我想对于使用者来说这也是大部分情况下的需求），同样mkdir也会创建父目录。

> 需要注意的是，del方法会删除目录而不判断其是否为空，这一方面方便了使用，另一方面也可能造成一些预想不到的后果（比如拼写错路径而删除不应该删除的目录），所以请谨慎使用此方法。

关于FileUtil中更多工具方法，请参阅API文档。

文件类型判断-FileTypeUtil

由来

在文件上传时，有时候我们需要判断文件类型。但是又不能简单的通过扩展名来判断（防止恶意脚本等通过上传到服务器上），于是我们需要在服务端通过读取文件的首部几个二进制位来判断常用的文件类型。

使用

这个工具类使用非常简单，通过调用FileTypeUtil.getType即可判断，这个方法同时提供众多的重载方法，用于读取不同的文件和流。

```
File file = FileUtil.file("d:/test.jpg");
String type = FileTypeUtil.getType(file);
//输出 jpg则说明确实为jpg文件
Console.log(type);
```

原理和局限性

这个类是通过读取文件流中前N个byte值来判断文件类型，在类中我们通过Map形式将常用的文件类型做了映射，这些映射都是网络上搜集而来。也就是说，我们只能识别有限的几种文件类型。但是这些类型已经涵盖了常用的图片、音频、视频、Office文档类型，可以应对大部分的使用场景。

> 对于某些文本格式的文件我们并不能通过首部byte判断其类型，比如JSON，这类文件本质上是文本文件，我们应该读取其文本内容，通过其语法判断类型。

自定义类型

为了提高FileTypeUtil的扩展性，我们通过putFileType方法可以自定义文件类型。

```
FileTypeUtil.putFileType("ffd8ffe000104a464946", "new_jpg");
```

第一个参数是文件流的前N个byte的16进制表示，我们可以读取自定义文件查看，选取一定长度即可(长度越长越精确)，第二个参数就是文件类型，然后使用FileTypeUtil.getType即可。

文件监听-WatchMonitor

由来

很多时候我们需要监听一个文件的变化或者目录的变动，包括文件的创建、修改、删除，以及目录下文件的创建、修改和删除，在JDK7前我们只能靠轮询方式遍历目录或者定时检查文件的修改事件，这样效率非常低，性能也很差。因此在JDK7中引入了WatchService。不过考虑到其API并不友好，于是Hutool便针对其做了简化封装，使监听更简单，也提供了更好的功能，这包括：

- 支持多级目录的监听（WatchService只支持一级目录），可自定义监听目录深度
- 延迟合并触发支持（文件变动时可能触发多次modify，支持在某个时间范围内的多次修改事件合并为一个修改事件）
- 简洁易懂的API方法，一个方法即可搞定监听，无需理解复杂的监听注册机制。
- 多观察者实现，可以根据业务实现多个Watcher来响应同一个事件（通过WatcherChain）

WatchMonitor

在Hutool中，WatchMonitor主要针对JDK7中WatchService做了封装，针对文件和目录的变动

（创建、更新、删除）做一个钩子，在Watcher中定义相应的逻辑来应对这些文件的变化。

内部应用

在hutool-setting模块，使用WatchMonitor监测配置文件变化，然后自动load到内存中。
WatchMonitor的使用可以避免轮询，以事件响应的方式应对文件变化。

使用

WatchMonitor提供的事件有：

- ENTRY_MODIFY 文件修改的事件
- ENTRY_CREATE 文件或目录创建的事件
- ENTRY_DELETE 文件或目录删除的事件
- OVERFLOW 丢失的事件

这些事件对应StandardWatchEventKinds中的事件。

下面我们介绍WatchMonitor的使用：

监听指定事件

```
File file = FileUtil.file("example.properties");
//这里只监听文件或目录的修改事件
WatchMonitor watchMonitor = WatchMonitor.create(file,
WatchMonitor.ENTRY_MODIFY);
watchMonitor.setWatcher(new Watcher(){
@Override
public void onCreate(WatchEvent<?> event, Path currentPath) {
Object obj = event.context();
Console.log("创建：{}-> {}", currentPath, obj);
}

@Override
public void onModify(WatchEvent<?> event, Path currentPath) {
Object obj = event.context();
Console.log("修改：{}-> {}", currentPath, obj);
}

@Override
public void onDelete(WatchEvent<?> event, Path currentPath) {
Object obj = event.context();
Console.log("删除：{}-> {}", currentPath, obj);
}
```

```
@Override
public void onOverflow(WatchEvent<?> event, Path currentPath) {
Object obj = event.context();
Console.log("Overflow：{}-> {}", currentPath, obj);
}
});

//设置监听目录的最大深入，目录层级大于制定层级的变更将不被监听，默认只监听当前层级目录
watchMonitor.setMaxDepth(3);
//启动监听
watchMonitor.start();
```

监听全部事件

其实我们不必实现Watcher的所有接口方法，Hutool同时提供了SimpleWatcher类，只需重写对应方法即可。

同样，如果我们想监听所有事件，可以：

```
WatchMonitor.createAll(file, new SimpleWatcher(){
@Override
public void onModify(WatchEvent<?> event, Path currentPath) {
Console.log("EVENT modify");
}
}).start();
```

createAll方法会创建一个监听所有事件的WatchMonitor，同时在第二个参数中定义Watcher来负责处理这些变动。

延迟处理监听事件

在监听目录或文件时，如果这个文件有修改操作，JDK会多次触发modify方法，为了解决这个问题，我们定义了DelayWatcher，此类通过维护一个Set将短时间内相同文件多次modify的事件合并处理触发，从而避免以上问题。

```
WatchMonitor monitor = WatchMonitor.createAll("d:/", new DelayWatcher(watcher, 500));
monitor.start();
```

ClassPath资源访问-ClassPathResource

什么是ClassPath

简单说来ClassPath就是查找class文件的路径，在Tomcat等容器下，ClassPath一般是WEB-INF/classes，在普通java程序中，我们可以通过定义-cp或者-classpath参数来定义查找class文件的路径，这些路径就是ClassPath。

为了项目方便，我们定义的配置文件肯定不能使用绝对路径，所以需要使用相对路径，这时候最好的办法就是把配置文件和class文件放在一起，便于查找。

由来

在Java编码过程中，我们常常希望读取项目内的配置文件，按照Maven的习惯，这些文件一般放在项目的src/main/resources下，读取的时候使用：

```
String path = "config.properties";
InputStream in = this.class.getResource(path).openStream();
```

使用当前类来获得资源其实就是使用当前类的类加载器获取资源，最后openStream()方法获取输入流来读取文件流。

封装

面对这种复杂的读取操作，我们封装了ClassPathResource类来简化这种资源的读取：

```
ClassPathResource resource = new ClassPathResource("test.properties");
Properties properties = new Properties();
properties.load(resource.getStream());

Console.log("Properties: {}", properties);
```

这样就大大简化了ClassPath中资源的读取。

> Hutool提供针对properties的封装类Props，同时提供更加强大的配置文件Setting类，这两个类已经针对ClassPath做过相应封装，可以以更加便捷的方式读取配置文件。相关文档请参阅Hutool-setting章节

文件读取-FileReader

由来

在FileUtil中本来已经针对文件的读操作做了大量的静态封装，但是根据职责分离原则，我觉得有必要针对文件读取单独封装一个类，这样项目更加清晰。当然，使用FileUtil操作文件是最方便的。

使用

在JDK中，同样有一个FileReader类，但是并不如想象中的那样好用，于是Hutool便提供了更加便捷FileReader类。

```
//默认UTF-8编码，可以在构造中传入第二个参数做为编码
FileReader fileReader = new FileReader("test.properties");
String result = fileReader.readString();
```

FileReader提供了以下方法来快速读取文件内容：

- readBytes
- readString
- readLines

同时，此类还提供了以下方法用于转换为流或者BufferedReader：

- getReader
- getInputStream

文件写入-FileWriter

相应的，文件读取有了，自然有文件写入类，使用方式与FileReader也类似：

```
FileWriter writer = new FileWriter("test.properties");
writer.write("test");
```

写入文件分为追加模式和覆盖模式两类，追加模式可以用append方法，覆盖模式可以用write方法，同时也提供了一个write方法，第二个参数是可选覆盖模式。

同样，此类提供了：

- getOutputStream
- getWriter
- getPrintWriter

这些方法用于转换为相应的类提供更加灵活的写入操作。

工具类

概述

包含内容

此包中的工具类为未经过分类的一些工具类，提供一些常用的工具方法。

此包中根据用途归类为XXXUtil，提供大量的工具方法。在工具类中，主要以类方法（static方法）为主，且各个类无法实例化为对象，一个方法是一个独立功能，无相互影响。

关于工具类的说明和使用，请参阅下面的章节。

数组工具-ArrayUtil

介绍

数组工具中的方法在2.x版本中都在CollectionUtil中存在，3.x版本中拆分出来作为ArrayUtil。数组工具类主要针对原始类型数组和泛型数组相关方案进行封装。

数组工具类主要是解决对象数组（包括包装类型数组）和原始类型数组使用方法不统一的问题。

方法

判空

数组的判空类似于字符串的判空，标准是null或者数组长度为0，ArrayUtil中封装了针对原始类型和泛型数组的判空和判非空：

1. 判断空

```
int[] a = {};
int[] b = null;
ArrayUtil.isEmpty(a);
ArrayUtil.isEmpty(b);
```

1. 判断非空

```
int[] a = {1,2};
ArrayUtil.isNotEmpty(a);
```

新建泛型数组

Array.newInstance并不支持泛型返回值，在此封装此方法使之支持泛型返回值。

```
String[] newArray = ArrayUtil.newArray(String.class, 3);
```

调整大小

使用 ArrayUtil.resize方法生成一个新的重新设置大小的数组。

合并数组

ArrayUtil.addAll方法采用可变参数方式，将多个泛型数组合并为一个数组。

克隆

数组本身支持clone方法，因此确定为某种类型数组时调用ArrayUtil.clone(T[]),不确定类型的使用`ArrayUtil.clone(T)`，两种重载方法在实现上有所不同，但是在使用中并不能感知出差别。

1. 泛型数组调用原生克隆

```
Integer[] b = {1,2,3};
Integer[] cloneB = ArrayUtil.clone(b);
Assert.assertArrayEquals(b, cloneB);
```

1. 非泛型数组（原始类型数组）调用第二种重载方法

```
int[] a = {1,2,3};
int[] clone = ArrayUtil.clone(a);
Assert.assertArrayEquals(a, clone);
```

有序列表生成

ArrayUtil.range方法有三个重载，这三个重载配合可以实现支持步进的有序数组或者步进为1的有序数组。这种列表生成器在Python中做为语法糖存在。

拆分数组

ArrayUtil.split方法用于拆分一个byte数组，将byte数组平均分成几等份，常用于消息拆分。

过滤

ArrayUtil.filter方法用于编辑已有数组元素，只针对泛型数组操作，原始类型数组并未提供。

方法中Editor接口用于返回每个元素编辑后的值，返回null此元素将被抛弃。

一个大栗子：过滤数组，只保留偶数

```
Integer[] a = {1,2,3,4,5,6};
Integer[] filter = ArrayUtil.filter(a, new Editor<Integer>(){
@Override
public Integer edit(Integer t) {
return (t % 2 == 0) ? t : null;
}});
Assert.assertArrayEquals(filter, new Integer[]{2,4,6});
```

zip

ArrayUtil.zip方法传入两个数组，第一个数组为key，第二个数组对应位置为value，此方法在Python中为zip()函数。

```
String[] keys = {"a", "b", "c"};
Integer[] values = {1,2,3};
Map<String, Integer> map = ArrayUtil.zip(keys, values, true);

//{a=1, b=2, c=3}
```

是否包含元素

ArrayUtil.contains方法只针对泛型数组，检测指定元素是否在数组中。

包装和拆包

在原始类型元素和包装类型中，Java实现了自动包装和拆包，但是相应的数组无法实现，于是便是用ArrayUtil.wrap和ArrayUtil.unwrap对原始类型数组和包装类型数组进行转换。

判断对象是否为数组

ArrayUtil.isArray方法封装了obj.getClass().isArray()。

转为字符串

1. ArrayUtil.toString 通常原始类型的数组输出为字符串时无法正常显示，于是封装此方法可以完美兼容原始类型数组和包装类型数组的转为字符串操作。

2. ArrayUtil.join 方法使用间隔符将一个数组转为字符串，比如[1,2,3,4]这个数组转为字符串，间隔符使用“-”的话，结果为 1-2-3-4，join方法同样支持泛型数组和原始类型数组。

toArray

ArrayUtil.toArray方法针对ByteBuffer转数组提供便利。

字符编码工具-CharsetUtil

介绍

CharsetUtil主要针对编码操作做了工具化封装，同时提供了一些常用编码常量。

常量

常量在需要编码的地方直接引用，可以很好的提高便利性。

字符串形式

1. ISO*8859*1

2. UTF_8

3. GBK

Charset对象形式

1. CHARSET*ISO*8859_1

2. CHARSET*UTF*8

3. CHARSET_GBK

方法

编码字符串转为Charset对象

CharsetUtil.charset方法用于将编码形式字符串转为Charset对象。

转换编码

CharsetUtil.convert方法主要是在两种编码中转换。主要针对因为编码识别错误而导致的乱码问题的一种解决方法。

系统默认编码

CharsetUtil.defaultCharset方法是Charset.defaultCharset()的封装方法。返回系统编码。

CharsetUtil.defaultCharsetName方法返回字符串形式的编码类型。

类工具-ClassUtil

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。

getShortClassName

获取完整类名的短格式如：cn.hutool.core.util.StrUtil -> c.h.c.u.StrUtil

isAllAssignableFrom

比较判断types1和types2两组类，如果types1中所有的类都与types2对应位置的类相同，或者是其父类或接口，则返回true

isPrimitiveWrapper

是否为包装类型

isBasicType

是否为基本类型（包括包装类和原始类）

getPackage

获得给定类所在包的名称，例如：

com.xiaoleilu.hutool.util.ClassUtil -> com.xiaoleilu.hutool.util

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。

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。

getJavaClassPaths方法

此方法用于获得java的系统变量定义的ClassPath。

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。

getDefaultValue

获取指定类型分的默认值，默认值规则为：

1. 如果为原始类型，返回0
2. 非原始类型返回null

其它

更多详细的方法描述见：

https://apidoc.gitee.com/loolly/hutool/cn/hutool/core/util/ClassUtil.html(https://apidoc.gitee.com/loolly/hutool/cn/hutool/core/util/ClassUtil.html)

类加载工具-ClassLoaderUtil

介绍

提供ClassLoader相关的工具类，例如类加载（Class.forName包装）等

方法

获取ClassLoader

getContextClassLoader

获取当前线程的ClassLoader，本质上调用Thread.currentThread().getContextClassLoader()

getClassLoader

按照以下顺序规则查找获取ClassLoader：

1. 获取当前线程的ContextClassLoader
2. 获取ClassLoaderUtil类对应的ClassLoader
3. 获取系统ClassLoader（ClassLoader.getSystemClassLoader()）

加载Class

loadClass

加载类，通过传入类的字符串，返回其对应的类名，使用默认ClassLoader并初始化类（调用static模块内容和可选的初始化static属性）

扩展Class.forName方法，支持以下几类类名的加载：

1. 原始类型，例如：int
2. 数组类型，例如：int[]、Long[]、String[]
3. 内部类，例如：java.lang.Thread.State会被转为java.lang.Thread$State加载

同时提供loadPrimitiveClass方法用于快速加载原始类型的类。包括原始类型、原始类型数组和void

isPresent

指定类是否被提供，通过调用loadClass方法尝试加载指定类名的类，如果加载失败返回false。

加载失败的原因可能是此类不存在或其关联引用类不存在。

Escape工具-EscapeUtil

介绍

转义和反转义工具类Escape / Unescape。escape采用ISO Latin字符集对指定的字符串进行编码。所有的空格符、标点符号、特殊字符以及其他非ASCII字符都将被转化成%xx格式的字符编码(xx等于该字符在字符集表里面的编码的16进制数字)。

此类中的方法对应Javascript中的escape()函数和unescape()函数。

方法

1. EscapeUtil.escape Escape编码（Unicode），该方法不会对 ASCII 字母和数字进行编码，也不会对下面这些 ASCII 标点符号进行编码： * @ - _ + . / 。其他所有的字符都会被转义序列替换。
2. EscapeUtil.unescape Escape解码。
3. EscapeUtil.safeUnescape 安全的unescape文本，当文本不是被escape的时候，返回原文。

16进制工具-HexUtil

介绍

十六进制（简写为hex或下标16）在数学中是一种逢16进1的进位制，一般用数字0到9和字母A到F表示（其中:A~F即10~15）。例如十进制数57，在二进制写作111001，在16进制写作39。

像java,c这样的语言为了区分十六进制和十进制数值,会在十六进制数的前面加上 0x,比如0x20是十进制的32,而不是十进制的20。HexUtil就是将字符串或byte数组与16进制表示转换的工具类。

用于

16进制一般针对无法显示的一些二进制进行显示，常用于：

1、图片的字符串表现形式

2、加密解密

3、编码转换

使用

HexUtil主要以encodeHex和decodeHex两个方法为核心，提供一些针对字符串的重载方法。

```
String str = "我是一个字符串";

String hex = HexUtil.encodeHexStr(str, CharsetUtil.CHARSET_UTF_8);

//hex是：
//e68891e698afe4b880e4b8aae5ad97e7aca6e4b8b2

String decodedStr = HexUtil.decodeHexStr(hex);

//解码后与str相同
```

Hash算法-HashUtil

介绍

HashUtil其实是一个hash算法的集合，此工具类中融合了各种hash算法。

方法

这些算法包括：

1. additiveHash 加法hash
2. rotatingHash 旋转hash
3. oneByOneHash 一次一个hash
4. bernstein Bernstein's hash
5. universal Universal Hashing
6. zobrist Zobrist Hashing
7. fnvHash 改进的32位FNV算法1
8. intHash Thomas Wang的算法，整数hash
9. rsHash RS算法hash
10. jsHash JS算法
11. pjwHash PJW算法
12. elfHash ELF算法
13. bkdrHash BKDR算法
14. sdbmHash SDBM算法
15. djbHash DJB算法
16. dekHash DEK算法
17. apHash AP算法
18. tianlHash TianL Hash算法
19. javaDefaultHash JAVA自己带的算法
20. mixHash 混合hash算法，输出64位的值

身份证工具-IdcardUtil

由来

在日常开发中，我们对身份证的验证主要是正则方式（位数，数字范围等），但是中国身份证，尤其18位身份证每一位都有严格规定，并且最后一位为校验位。而我们在实际应用中，针对身份证的验证理应严格至此。于是IdcardUtil应运而生。

> IdcardUtil从3.0.4版本起加入Hutool工具家族，请升级至此版本以上可使用。

介绍

IdcardUtil现在支持大陆15位、18位身份证，港澳台10位身份证。

工具中主要的方法包括：

1. isValidCard 验证身份证是否合法
2. convert15To18 身份证15位转18位
3. getBirthByIdCard 获取生日
4. getAgeByIdCard 获取年龄
5. getYearByIdCard 获取生日年
6. getMonthByIdCard 获取生日月
7. getDayByIdCard 获取生日天
8. getGenderByIdCard 获取性别
9. getProvinceByIdCard 获取省份

使用

```
String ID_18 = "321083197812162119";
String ID_15 = "150102880730303";

//是否有效
boolean valid = IdcardUtil.isValidCard(ID_18);
boolean valid15 = IdcardUtil.isValidCard(ID_15);

//转换
String convert15To18 = IdcardUtil.convert15To18(ID_15);
Assert.assertEquals(convert15To18, "150102198807303035");

//年龄
DateTime date = DateUtil.parse("2017-04-10");

int age = IdcardUtil.getAgeByIdCard(ID_18, date);
Assert.assertEquals(age, 38);

int age2 = IdcardUtil.getAgeByIdCard(ID_15, date);
Assert.assertEquals(age2, 28);

//生日
String birth = IdcardUtil.getBirthByIdCard(ID_18);
Assert.assertEquals(birth, "19781216");

String birth2 = IdcardUtil.getBirthByIdCard(ID_15);
Assert.assertEquals(birth2, "19880730");

//省份
```

```
String province = IdcardUtil.getProvinceByIdCard(ID_18);
Assert.assertEquals(province, "江苏");

String province2 = IdcardUtil.getProvinceByIdCard(ID_15);
Assert.assertEquals(province2, "内蒙古");
```

> **声明**
>
> 以上两个身份证号码为随机编造的，如有雷同，纯属巧合。

图片工具-ImageUtil

介绍

针对awt中图片处理进行封装，这些封装包括：缩放、裁剪、转为黑白、加水印等操作。

方法介绍

1. scale 缩放图片，提供两种重载方法：其中一个是按照长宽缩放，另一种是按照比例缩放。
2. cut 剪裁图片
3. cutByRowsAndCols 按照行列剪裁（将图片分为20行和20列）
4. convert 图片类型转换，支持GIF->JPG、GIF->PNG、PNG->JPG、PNG->GIF(X)、BMP->PNG等
5. gray 彩色转为黑白
6. pressText 添加文字水印
7. pressImage 添加图片水印
8. rotate 旋转图片
9. flip 水平翻转图片

数字工具-NumberUtil

由来

数字工具针对数学运算做工具性封装

使用

加减乘除

- NumberUtil.add 针对double类型做加法
- NumberUtil.sub 针对double类型做减法

• NumberUtil.mul 针对double类型做乘法

• NumberUtil.div 针对double类型做除法，并提供重载方法用于规定除不尽的情况下保留小数位数和舍弃方式。

以上四种运算都会将double转为BigDecimal后计算，解决float和double类型无法进行精确计算的问题。这些方法常用于商业计算。

保留小数

保留小数的方法主要有两种：

• NumberUtil.round 方法主要封装BigDecimal中的方法来保留小数，返回double，这个方法更加灵活，可以选择四舍五入或者全部舍弃等模式。

```
double te1=123456.123456;
double te2=123456.128456;
Console.log(round(te1,4));//结果:123456.12
Console.log(round(te2,4));//结果:123456.13
```

• NumberUtil.roundStr 方法主要封装String.format方法,舍弃方式采用四舍五入。

```
double te1=123456.123456;
double te2=123456.128456;
Console.log(roundStr(te1,2));//结果:123456.12
Console.log(roundStr(te2,2));//结果:123456.13
```

decimalFormat

针对 DecimalFormat.format进行简单封装。按照固定格式对double或long类型的数字做格式化操作。

```
long c=299792458;//光速
String format = NumberUtil.decimalFormat(",###", c);//299,792,458
```

格式中主要以 # 和 0 两种占位符号来指定数字长度。0 表示如果位数不足则以 0 填充，# 表示只要有可能就把数字拉上这个位置。

• 0 -> 取一位整数

• 0.00 -> 取一位整数和两位小数

• 00.000 -> 取两位整数和三位小数

• # -> 取所有整数部分

• #.##% -> 以百分比方式计数，并取两位小数

• #.#####E0 -> 显示为科学计数法，并取五位小数

• ,### -> 每三位以逗号进行分隔，例如：299,792,458

• 光速大小为每秒,###米 -> 将格式嵌入文本

关于格式的更多说明，请参阅：Java DecimalFormat的主要功能及使用方法

(http://blog.csdn.net/evangel_z/article/details/7624503)

是否为数字

- NumberUtil.isNumber 是否为数字
- NumberUtil.isInteger 是否为整数
- NumberUtil.isDouble 是否为浮点数
- NumberUtil.isPrimes 是否为质数

随机数

- NumberUtil.generateRandomNumber 生成不重复随机数 根据给定的最小数字和最大数字，以及随机数的个数，产生指定的不重复的数组。
- NumberUtil.generateBySet 生成不重复随机数 根据给定的最小数字和最大数字，以及随机数的个数，产生指定的不重复的数组。

整数列表

NumberUtil.range 方法根据范围和步进，生成一个有序整数列表。

NumberUtil.appendRange 将给定范围内的整数添加到已有集合中

其它

- NumberUtil.factorial 阶乘
- NumberUtil.sqrt 平方根
- NumberUtil.divisor 最大公约数
- NumberUtil.multiple 最小公倍数
- NumberUtil.getBinaryStr 获得数字对应的二进制字符串
- NumberUtil.binaryToInt 二进制转int
- NumberUtil.binaryToLong 二进制转long
- NumberUtil.compare 比较两个值的大小
- NumberUtil.toStr 数字转字符串，自动并去除尾小数点儿后多余的0

网络工具-NetUtil

由来

在日常开发中，网络连接这块儿必不可少。日常用到的一些功能,隐藏掉部分IP地址、绝对相对路径的转换等等。

介绍

NetUtil 工具中主要的方法包括：

1. longToIpv4 根据long值获取ip v4地址
2. ipv4ToLong 根据ip地址计算出long型的数据
3. isUsableLocalPort 检测本地端口可用性
4. isValidPort 是否为有效的端口
5. isInnerIP 判定是否为内网IP
6. localIpv4s 获得本机的IP地址列表
7. toAbsoluteUrl 相对URL转换为绝对URL
8. hideIpPart 隐藏掉IP地址的最后一部分为 * 代替
9. buildInetSocketAddress 构建InetSocketAddress
10. getIpByHost 通过域名得到IP
11. isInner 指定IP的long是否在指定范围内

使用

```
String ip= "127.0.0.1";
long iplong = 2130706433L;

//根据long值获取ip v4地址
String ip= NetUtil.longToIpv4(iplong);


//根据ip地址计算出long型的数据
long ip= NetUtil.ipv4ToLong(ip);

//检测本地端口可用性
boolean result= NetUtil.isUsableLocalPort(6379);

//是否为有效的端口
boolean result= NetUtil.isValidPort(6379);

//隐藏掉IP地址
 String result =NetUtil.hideIpPart(ip);
```

更多方法请见：

API文档-NetUtil(https://apidoc.gitee.com/loolly/hutool/cn/hutool/core/util/NetUtil.html)

分页工具-PageUtil

由来

分页工具类并不是数据库分页的封装，而是分页方式的转换。在我们手动分页的时候，常常使用页码+每页个数的方式，但是有些数据库需要使用开始位置和结束位置来表示。很多时候这种转换容易出错（边界问题），于是封装了PageUtil工具类。

使用

transToStartEnd

将页数和每页条目数转换为开始位置和结束位置。

此方法用于不包括结束位置的分页方法。

例如：

- 页码：1，每页10 -> [0, 10]
- 页码：2，每页10 -> [10, 20]

```
int[] startEnd1 = PageUtil.transToStartEnd(1, 10);//[0, 10]
int[] startEnd2 = PageUtil.transToStartEnd(2, 10);//[10, 20]
```

> 方法中，页码从1开始，位置从0开始

totalPage

根据总数计算总页数

```
int totalPage = PageUtil.totalPage(20, 3);//7
```

分页彩虹算法

此方法来自：https://github.com/iceroot/iceroot/blob/master/src/main/java/com/icexxx/util/IceUtil.java

在页面上显示下一页时，常常需要显示前N页和后N页，PageUtil.rainbow作用于此。

例如我们当前页为第5页，共有20页，只显示6个页码，显示的分页列表应为：

```
上一页 3 4 [5] 6 7 8 下一页
```

```
//参数意义分别为：当前页、总页数、每屏展示的页数
int[] rainbow = PageUtil.rainbow(5, 20, 6);
//结果：[3, 4, 5, 6, 7, 8]
```

随机工具-RandomUtil

说明

RandomUtil主要针对JDK中Random对象做封装，严格来说，Java产生的随机数都是伪随机数，因此Hutool封装后产生的随机结果也是伪随机结果。不过这种随机结果对于大多数情况已经够用。

使用

- RandomUtil.randomInt 获得指定范围内的随机数
- RandomUtil.randomBytes 随机bytes
- RandomUtil.randomEle 随机获得列表中的元素
- RandomUtil.randomEleSet 随机获得列表中的一定量的不重复元素，返回Set
- RandomUtil.randomString 获得一个随机的字符串（只包含数字和字符）
- RandomUtil.randomNumbers 获得一个只包含数字的字符串
- RandomUtil.randomUUID 随机UUID
- RandomUtil.weightRandom 权重随机生成器，传入带权重的对象，然后根据权重随机获取对象

对象工具-ObjectUtil

由来

在我们的日常使用中，有些方法是针对Object通用的，这些方法不区分何种对象，针对这些方法，Hutool封装为ObjectUtil。

方法

ObjectUtil.equal

比较两个对象是否相等，相等需满足以下条件之一：

1. obj1 == null && obj2 == null

2. obj1.equals(obj2)

ObjectUtil.length

计算对象长度，如果是字符串调用其length方法，集合类调用其size方法，数组调用其length属性，其他可遍历对象遍历计算长度。

支持的类型包括：

- CharSequence
- Collection
- Map
- Iterator
- Enumeration

• Array

ObjectUtil.contains

对象中是否包含元素。

支持的对象类型包括：

• String
• Collection
• Map
• Iterator
• Enumeration
• Array

判断是否为null

• ObjectUtil.isNull
• ObjectUtil.isNotNull

克隆

• ObjectUtil.clone 克隆对象，如果对象实现Cloneable接口，调用其clone方法，如果实现Serializable接口，执行深度克隆，否则返回null。
• ObjectUtil.cloneIfPossible 返回克隆后的对象，如果克隆失败，返回原对象
• ObjectUtil.cloneByStream 序列化后拷贝流的方式克隆，对象必须实现Serializable接口

序列化和反序列化

• serialize 序列化，调用JDK序列化
• unserialize 反序列化，调用JDK

判断基本类型

ObjectUtil.isBasicType 判断是否为基本类型，包括包装类型和非包装类型。

字符串工具-StrUtil

由来

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。

方法

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. str、bytes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……）

Java

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。

更多方法请参阅API文档。

正则工具-ReUtil

由来

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);
    }
```

使用

ReUtil.extractMulti

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");
Assert.assertEquals("Z-a", resultExtractMulti);
```

ReUtil.delFirst

删除第一个匹配到的内容

```
String resultDelFirst = ReUtil.delFirst("(\\w)aa(\\w)", content);
Assert.assertEquals("ZZbbbccc中文1234", resultDelFirst);
```

ReUtil.findAll

查找所有匹配文本

```
List<String> resultFindAll = ReUtil.findAll("\\w{2}", content, 0, new
ArrayList<String>());
ArrayList<String> expected = CollectionUtil.newArrayList("ZZ", "Za", "aa", "bb", "bc",
"cc", "12", "34");
Assert.assertEquals(expected, resultFindAll);
```

ReUtil.getFirstNumber

找到匹配的第一个数字

```
Integer resultGetFirstNumber = ReUtil.getFirstNumber(content);
```

```
Assert.assertEquals(Integer.valueOf(1234), resultGetFirstNumber);
```

ReUtil.isMatch

给定字符串是否匹配给定正则

```
boolean isMatch = ReUtil.isMatch("\\w+[\u4E00-\u9FFF]+\\d+", content);
Assert.assertTrue(isMatch);
```

ReUtil.replaceAll

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

```
//此处把1234替换为 ->1234<-
String replaceAll = ReUtil.replaceAll(content, "(\\d+)", "->$1<-");
Assert.assertEquals("ZZZaaabbbccc中文->1234<-", replaceAll);
```

ReUtil.escape

转义给定字符串，为正则相关的特殊符号转义

```
String escape = ReUtil.escape("我有个$符号{}");
Assert.assertEquals("我有个\\$符号\\{\\}", escape);
```

URL工具-URLUtil

介绍

URL（Uniform Resource Locator）中文名为统一资源定位符，有时也被俗称为网页地址。表示为互联网上的资源，如网页或者FTP地址。在Java中，也可以使用URL表示Classpath中的资源（Resource）地址。

方法

获取URL对象

- URLUtil.url 通过一个字符串形式的URL地址创建对象
- URLUtil.getURL 主要获得ClassPath下资源的URL，方便读取Classpath下的配置文件等信息。

其它

- URLUtil.formatUrl 格式化URL链接。对于不带http://头的地址做简单补全。
- URLUtil.encode 封装URLEncoder.encode，将需要转换的内容（ASCII码形式之外的内容），用十六进制表示法转换出来，并在之前加上%开头。
- URLUtil.decode 封装URLDecoder.decode，将%开头的16进制表示的内容解码。
- URLUtil.getPath 获得path部分 URI -> http://www.aaa.bbb/search?scope=ccc&q=ddd PATH -> /search
- URLUtil.toURI 转URL或URL字符串为URI。

XML工具-XmlUtil

由来

在日常编码中，我们接触最多的除了JSON外，就是XML格式了，一般而言，我们首先想到的是引入Dom4j包，却不知JDK已经封装有XML解析和构建工具：w3c dom。但是由于这个API操作比较繁琐，因此Hutool中提供了XmlUtil简化XML的创建、读和写的过程。

使用

读取XML

读取XML分为两个方法：

- XmlUtil.readXML 读取XML文件
- XmlUtil.parseXml 解析XML字符串为Document对象

写XML

- XmlUtil.toStr 将XML文档转换为String
- XmlUtil.toFile 将XML文档写入到文件

创建XML

- XmlUtil.createXml 创建XML文档, 创建的XML默认是utf8编码，修改编码的过程是在toStr和toFile方法里，既XML在转为文本的时候才定义编码。

XML操作

通过以下工具方法，可以完成基本的节点读取操作。

- XmlUtil.cleanInvalid 除XML文本中的无效字符
- XmlUtil.getElements 根据节点名获得子节点列表
- XmlUtil.getElement 根据节点名获得第一个子节点
- XmlUtil.elementText 根据节点名获得第一个子节点

- XmlUtil.transElements 将NodeList转换为Element列表

XML与对象转换

- writeObjectAsXml 将可序列化的对象转换为XML写入文件，已经存在的文件将被覆盖。
- readObjectFromXml 从XML中读取对象。

> 注意
>
> 这两个方法严重依赖JDK的XMLEncoder和XMLDecoder，生成和解析必须成对存在（遵循固定格式），普通的XML转Bean会报错。

Xpath操作

Xpath的更多介绍请看文章：https://www.ibm.com/developerworks/cn/xml/x-javaxpathapi.html(https://www.ibm.com/developerworks/cn/xml/x-javaxpathapi.html)

- createXPath 创建XPath
- getByXPath 通过XPath方式读取XML节点等信息

栗子：

```
<?xml version="1.0" encoding="utf-8"?>

<returnsms>
 <returnstatus>Success（成功）</returnstatus>
 <message>ok</message>
 <remainpoint>1490</remainpoint>
 <taskID>885</taskID>
 <successCounts>1</successCounts>
</returnsms>
```

```
Document docResult=XmlUtil.readXML(xmlFile);
//结果为“ok”
Object value = XmlUtil.getByXPath("//returnsms/message", docResult,
XPathConstants.STRING);
```

总结

XmlUtil只是w3c dom的简单工具化封装，减少操作dom的难度，如果项目对XML依赖较大，依旧推荐Dom4j框架。

压缩工具-ZipUtil

由来

在Java中，对文件、文件夹打包，压缩是一件比较繁琐的事情，我们常常引入Zip4j进行此类操作。但是很多时候，JDK中的zip包就可满足我们大部分需求。ZipUtil就是针对java.util.zip做工具化封装，使压缩解压操作可以一个方法搞定，并且自动处理文件和目录的问题，不再需要用户判断，压缩后的文件也会自动创建文件，自动创建父目录，大大简化的压缩解压的复杂度。

方法

Zip

1. 压缩

ZipUtil.zip 方法提供一系列的重载方法，满足不同需求的压缩需求，这包括：

• 打包到当前目录（可以打包文件，也可以打包文件夹，根据路径自动判断）

```
//将aaa目录下的所有文件目录打包到d:/aaa.zip
ZipUtil.zip("d:/aaa");
```

• 指定打包后保存的目的地，自动判断目标是文件还是文件夹

```
//将aaa目录下的所有文件目录打包到d:/bbb/目录下的aaa.zip文件中
ZipUtil.zip("d:/aaa", "d:/bbb/");

//将aaa目录下的所有文件目录打包到d:/bbb/目录下的ccc.zip文件中
ZipUtil.zip("d:/aaa", "d:/bbb/ccc.zip");
```

• 可选是否包含被打包的目录。比如我们打包一个照片的目录，打开这个压缩包有可能是带目录的，也有可能是打开压缩包直接看到的是文件。zip方法增加一个boolean参数可选这两种模式，以应对众多需求。

```
//将aaa目录以及其目录下的所有文件目录打包到d:/bbb/目录下的ccc.zip文件中
ZipUtil.zip("d:/aaa", "d:/bbb/ccc.zip", true);
```

• 多文件或目录压缩。可以选择多个文件或目录一起打成zip包。

```
ZipUtil.zip(FileUtil.file("d:/bbb/ccc.zip"), false,
    FileUtil.file("d:/test1/file1.txt"),
    FileUtil.file("d:/test1/file2.txt"),
    FileUtil.file("d:/test2/file1.txt"),
    FileUtil.file("d:/test2/file2.txt"),
);
```

1. 解压

ZipUtil.unzip 解压。同样提供几个重载，满足不同需求。

```
//将test.zip解压到e:\\aaa目录下，返回解压到的目录
File unzip = ZipUtil.unzip("E:\\aaa\\test.zip", "e:\\aaa");
```

Gzip

Gzip是网页传输中广泛使用的压缩方式，Hutool同样提供其工具方法简化其过程。

ZipUtil.gzip 压缩，可压缩字符串，也可压缩文件

ZipUtil.unGzip 解压Gzip文件

Zlib

ZipUtil.zlib 压缩，可压缩字符串，也可压缩文件

ZipUtil.unZlib 解压zlib文件

> 注意
>
> ZipUtil默认情况下使用系统编码，也就是说：
>
> 1. 如果你在命令行下运行，则调用系统编码（一般Windows下为GBK、Linux下为UTF-8）
> 2. 如果你在IDE（如Eclipse）下运行代码，则读取的是当前项目的编码（详细请查阅IDE设置，我的项目默认都是UTF-8编码，因此解压和压缩都是用这个编码）

反射工具-ReflectUtil

介绍

Java的反射机制，可以让语言变得更加灵活，对对象的操作也更加“动态”，因此在某些情况下，反射可以做到事半功倍的效果。Hutool针对Java的反射机制做了工具化封装，封装包括：

1. 获取构造方法
2. 获取字段
3. 获取字段值
4. 获取方法
5. 执行方法（对象方法和静态方法）

使用

获取某个类的所有方法

```
Method[] methods = ReflectUtil.getMethods(ExamInfoDict.class);
```

获取某个类的指定方法

```
Method method = ReflectUtil.getMethod(ExamInfoDict.class, "getId");
```

构造对象

```
ReflectUtil.newInstance(ExamInfoDict.class);
```

执行方法

```
class TestClass {
private int a;

public int getA() {
return a;
}

public void setA(int a) {
this.a = a;
}
}
```

```
TestClass testClass = new TestClass();
ReflectUtil.invoke(testClass, "setA", 10);
```

命令行工具-RuntimeUtil

介绍

在Java世界中，如果想与其它语言打交道，处理调用接口，或者JNI，就是通过本地命令方式调用了。Hutool封装了JDK的Process类，用于执行命令行命令（在Windows下是cmd，在Linux下是shell命令）。

方法

基础方法

1. exec 执行命令行命令，返回Process对象，Process可以读取执行命令后的返回内容的流

快捷方法

1. execForStr 执行系统命令，返回字符串
2. execForLines 执行系统命令，返回行列表

使用

```
String str = RuntimeUtil.execForStr("ipconfig");
```

执行这个命令后，在Windows下可以获取网卡信息。

剪贴板工具-ClipboardUtil

介绍

在Hutool群友的强烈要求下，在3.2.0+ 中新增了ClipboardUtil这个类用于简化操作剪贴板（当然使用场景被局限）。

使用

ClipboardUtil 封装了几个常用的静态方法:

通用方法

1. getClipboard 获取系统剪贴板
2. set 设置内容到剪贴板
3. get 获取剪贴板内容

针对文本

1. setStr 设置文本到剪贴板
2. getStr 从剪贴板获取文本

针对Image对象（图片）

1. setImage 设置图片到剪贴板
2. getImage 从剪贴板获取图片

枚举工具-EnumUtil

介绍

枚举（enum）算一种“语法糖”，是指一个经过排序的、被打包成一个单一实体的项列表。一个枚举的实例可以使用枚举项列表中任意单一项的值。枚举在各个语言当中都有着广泛的应用，通常用来表示诸如颜色、方式、类别、状态等等数目有限、形式离散、表达又极为明确的量。Java从JDK5开始，引入了对枚举的支持。

EnumUtil 用于对未知枚举类型进行操作。

方法

首先我们定义一个枚举对象：

```
//定义枚举
public enum TestEnum{
TEST1("type1"), TEST2("type2"), TEST3("type3");

private TestEnum(String type) {
this.type = type;
}

private String type;

public String getType() {
return this.type;
}
}
```

getNames

获取枚举类中所有枚举对象的name列表。栗子：

```
//定义枚举
public enum TestEnum {
TEST1, TEST2, TEST3;
}
```

```
List<String> names = EnumUtil.getNames(TestEnum.class);
//结果：[TEST1, TEST2, TEST3]
```

getFieldValues

获得枚举类中各枚举对象下指定字段的值。栗子：

```
List<Object> types = EnumUtil.getFieldValues(TestEnum.class, "type");
//结果：[type1, type2, type3]
```

getEnumMap

获取枚举字符串值和枚举对象的Map对应，使用LinkedHashMap保证有序，结果中键为枚举名，值为枚举对象。栗子：

```
Map<String,TestEnum> enumMap = EnumUtil.getEnumMap(TestEnum.class);
enumMap.get("TEST1") // 结果为：TestEnum.TEST1
```

getNameFieldMap

获得枚举名对应指定字段值的Map，键为枚举名，值为字段值。栗子：

```
Map<String, Object> enumMap = EnumUtil.getNameFieldMap(TestEnum.class,
"type");
enumMap.get("TEST1") // 结果为：type1
```

引用工具-ReferenceUtil

介绍

引用工具类，主要针对Reference 工具化封装

主要封装包括：

1. SoftReference 软引用，在GC报告内存不足时会被GC回收

2. WeakReference 弱引用，在GC时发现弱引用会回收其对象

3. PhantomReference 虚引用，在GC时发现虚引用对象，会将PhantomReference插入ReferenceQueue。此时对象未被真正回收，要等到ReferenceQueue被真正处理后才会被回收。

方法

create

根据类型枚举创建引用。

泛型类型工具-TypeUtil

介绍

针对 java.lang.reflect.Type 的工具类封装，最主要功能包括：

1. 获取方法的参数和返回值类型（包括Type和Class）

2. 获取泛型参数类型（包括对象的泛型参数或集合元素的泛型类型）

方法

首先我们定义一个类：

```
public class TestClass {
public List<String> getList(){
return new ArrayList<>();
}
```

```
public Integer intTest(Integer integer) {
return 1;
}
}
```

getClass

获得Type对应的原始类

getParamType

```
Method method = ReflectUtil.getMethod(TestClass.class, "intTest", Integer.class);
Type type = TypeUtil.getParamType(method, 0);
// 结果：Integer.class
```

获取方法参数的泛型类型

getReturnType

获取方法的返回值类型

```
Method method = ReflectUtil.getMethod(TestClass.class, "getList");
Type type = TypeUtil.getReturnType(method);
// 结果：java.util.List<java.lang.String>
```

getTypeArgument

获取泛型类子类中泛型的填充类型。

```
Method method = ReflectUtil.getMethod(TestClass.class, "getList");
Type type = TypeUtil.getReturnType(method);

Type type2 = TypeUtil.getTypeArgument(type);
// 结果：String.class
```

唯一ID工具-IdUtil

介绍

在分布式环境中，唯一ID生成应用十分广泛，生成方法也多种多样，Hutool针对一些常用生成策略做了简单封装。

唯一ID生成器的工具类，涵盖了：

- UUID
- ObjectId（MongoDB）
- Snowflake（Twitter）

使用

UUID

UUID全称通用唯一识别码（universally unique identifier），JDK通过java.util.UUID提供了Leach-Salz 变体的封装。在Hutool中，生成一个UUID字符串方法如下：

```
//生成的UUID是带-的字符串，类似于：a5c8a5e8-df2b-4706-bea4-08d0939410e3
String uuid = IdUtil.randomUUID();

//生成的是不带-的字符串，类似于：b17f24ff026d40949c85a24f4f375d42
String simpleUUID = IdUtil.simpleUUID();
```

> 说明
>
> Hutool重写java.util.UUID的逻辑，对应类为cn.hutool.core.lang.UUID，使生成不带-的UUID字符串不再需要做字符替换，性能提升一倍左右。

ObjectId

ObjectId是MongoDB数据库的一种唯一ID生成策略，是UUID version1的变种，详细介绍可见：服务化框架 - 分布式Unique ID的生成方法一览(http://calvin1978.blogcn.com/articles/uuid.html)。

Hutool针对此封装了cn.hutool.core.lang.ObjectId，快捷创建方法为：

```
//生成类似：5b9e306a4df4f8c54a39fb0c
String id = ObjectId.next();

//方法2：从Hutool-4.1.14开始提供
String id2 = IdUtil.objectId();
```

Snowflake

分布式系统中，有一些需要使用全局唯一ID的场景，有些时候我们希望能使用一种简单一些的ID，并且希望ID能够按照时间有序生成。Twitter的Snowflake 算法就是这种生成器。

使用方法如下：

```
//参数1为终端ID
```

```
//参数2为数据中心ID
Snowflake snowflake = IdUtil.createSnowflake(1, 1);
long id = snowflake.nextId();
```

语言特性

单例工具-Singleton

为什么会有这个类

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。

有界优先队列-BoundedPriorityQueue

简介

举个例子。我有一个用户表，这个表根据用户名被Hash到不同的数据库实例上，我要找出这些用户中最热门的5个，怎么做？我是这么做的：

1. 在每个数据库实例上找出最热门的5个

2. 将每个数据库实例上的这5条数据按照热门程度排序，最后取出前5条

这个过程看似简单，但是你应用服务器上的代码要写不少。首先需要Query N个列表，加入到一个新列表中，排序，再取前5。这个过程不但代码繁琐，而且牵涉到多个列表，非常浪费空间。

于是，BoundedPriorityQueue应运而生。

先看Demo：

```
/**
 * 有界优先队列Demo
 * @author Looly
 *
 */
public class BoundedPriorityQueueDemo {

public static void main(String[] args) {
//初始化队列，设置队列的容量为5（只能容纳5个元素），元素类型为integer使用默认比较器，在队列内部将按照从小到大排序
BoundedPriorityQueue<Integer> queue = new BoundedPriorityQueue<Integer>(5);

		//初始化队列，使用自定义的比较器
queue = new BoundedPriorityQueue<>(5, new Comparator<Integer>(){

@Override
public int compare(Integer o1, Integer o2) {
return o1.compareTo(o2);
}
});

//定义了6个元素，当元素加入到队列中，会按照从小到大排序，当加入第6个元素的时候，队列末尾（最大的元素）将会被抛弃
int[] array = new int[]{5,7,9,2,3,8};
for (int i : array) {
```

```
queue.offer(i);
}

            //队列可以转换为List哦~~
ArrayList<Integer> list = queue.toList();

System.out.println(queue);
}
}
```

原理非常简单。设定好队列的容量，然后把所有的数据add或者offer进去（两个方法相同），就会得到前5条数据了。

字段验证器-Validator

作用

验证给定字符串是否满足指定条件，一般用在表单字段验证里。

此类中全部为静态方法。

使用

判断验证

直接调用Validator.isXXX(String value)既可验证字段，返回是否通过验证。

例如：

```
boolean isEmail = Validator.isEmail("loolly@gmail.com")
```

表示验证给定字符串是否复合电子邮件格式。

其他验证信息请参阅Validator类

如果Validator里的方法无法满足自己的需求，那还可以调用

```
Validator.isByRegex("需要验证字段的正则表达式", "被验证内容")
```

来通过正则表达式灵活的验证内容。

异常验证

除了手动判断，我们有时需要在判断未满足条件时抛出一个异常，Validator同样提供异常验证机制：

```
Validator.validateChinese("我是一段zhongwen", "内容中包含非中文");
```

因为内容中包含非中文字符，因此会抛出ValidateException。

控制台打印封装-Console

由来

编码中我们常常需要调试输出一些信息，除了打印日志，最长用的要数System.out和System.err

比如我们打印一个Hello World，可以这样写：

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

但是面对纷杂的打印需求，System.out.println无法满足，比如：

1. 不支持参数，对象打印需要拼接字符串

2. 不能直接打印数组，需要手动调用Arrays.toString

考虑到以上问题，我封装了Console对象。

> Console对象的使用更加类似于Javascript的console.log()方法，这也是借鉴了JS的一个语法糖。

使用

1. Console.log 这个方法基本等同于System.out.println,但是支持类似于Slf4j的字符串模板语法，同时也会自动将对象（包括数组）转为字符串形式。

```
String[] a = {"abc", "bcd", "def"};
Console.log(a);//控制台输出：[abc, bcd, def]
```

```
Console.log("This is Console log for {}.", "test");
//控制台输出：This is Console log for test.
```

1. Console.error 这个方法基本等同于System.err.println，,但是支持类似于Slf4j的字符串模板语法，同时也会自动将对象（包括数组）转为字符串形式。

二进码十进数-BCD

介绍

BCD码（Binary-Coded Decimal）亦称二进码十进数或二-十进制代码，BCD码这种编码形式利用了四个位元来储存一个十进制的数码，使二进制和十进制之间的转换得以快捷的进行。

这种编码技巧最常用于会计系统的设计里，因为会计制度经常需要对很长的数字串作准确的计算。相对于一般的浮点式记数法，采用BCD码，既可保存数值的精确度，又可免却使电脑作浮点运算时所耗费的时间。此外，对于其他需要高精确度的计算，BCD编码亦很常用。

BCD码是四位二进制码, 也就是将十进制的数字转化为二进制, 但是和普通的转化有一点不同, 每一个十进制的数字0-9都对应着一个四位的二进制码,对应关系如下: 十进制0 对应 二进制0000 ;十进制

1 对应二进制0001 ..... 9 1001 接下来的10就有两个上述的码来表示 10 表示为00010000 也就是BCD码是遇见1001就产生进位,不象普通的二进制码,到1111才产生进位10000

方法

```
String strForTest = "123456ABCDEF";

//转BCD
byte[] bcd = BCD.strToBcd(strForTest);
//解码BCD
String str = BCD.bcdToStr(bcd);
Assert.assertEquals(strForTest, str);
```

HashMap扩展-Dict

由来

如果你了解Python，你一定知道Python有dict这一数据结构，也是一种KV（Key-Value）结构的数据结构，类似于Java中的Map，但是提供了更加灵活多样的使用。Hutool中的Dict对象旨在实现更加灵活的KV结构，针对强类型，提供丰富的getXXX操作，将HashMap扩展为无类型区别的数据结构。

介绍

Dict继承HashMap，其key为String类型，value为Object类型，通过实现BasicTypeGetter接口提供针对不同类型的get方法，同时提供针对Bean的转换方法，大大提高Map的灵活性。

> Hutool-db中Entity是Dict子类，做为数据的媒介。

使用

创建

```
Dict dict = Dict.create()
.set("key1", 1)//int
.set("key2", 1000L)//long
.set("key3", DateTime.now());//Date
```

通过链式构造，创建Dict对象，同时可以按照Map的方式使用。

获取指定类型的值

```
Long v2 = dict.getLong("key2");//1000
```

字符串格式化-StrFormatter

由来

我一直对Slf4j的字符串格式化情有独钟，通过{}这种简单的占位符完成字符串的格式化。于是参考Slf4j的源码，便有了StrFormatter。

> StrFormatter.format的快捷使用方式为StrUtil.format，推荐使用后者。

使用

```
//通常使用
String result1 = StrFormatter.format("this is {} for {}", "a", "b");
Assert.assertEquals("this is a for b", result1);

//转义{}
String result2 = StrFormatter.format("this is \\{} for {}", "a", "b");
Assert.assertEquals("this is {} for a", result2);

//转义\
String result3 = StrFormatter.format("this is \\\\{} for {}", "a", "b");
Assert.assertEquals("this is \\a for b", result3);
```

字符串切割-StrSpliter

由来

在Java的String对象中提供了split方法用于通过某种字符串分隔符来把一个字符串分割为数组。但是有的时候我们对这种操作有不同的要求，默认方法无法满足，这包括：

- 分割限制分割数
- 分割后每个字符串是否需要去掉两端空格
- 是否忽略空白片
- 根据固定长度分割
- 通过正则分隔

因此，StrSpliter应运而生。StrSpliter中全部为静态方法，方便快捷调用。

方法

基础方法

split 切分字符串，众多可选参数，返回结果为List

splitToArray 切分字符串，返回结果为数组

splitsplitByRegex 根据正则切分字符串

splitByLength 根据固定长度切分字符串

栗子：

```
String str1 = "a, ,efedsfs,   ddf";
//参数：被切分字符串，分隔符逗号，0表示无限制分片数，去除两边空格，忽略空白项
List<String> split = StrSpliter.split(str1, ',', 0, true, true);
```

特殊方法

splitPath 切分字符串，分隔符为"/"

splitPathToArray 切分字符串，分隔符为"/"，返回数组

断言-Assert

由来

Java中有assert关键字，但是存在许多问题：

1. assert关键字需要在运行时候显式开启才能生效，否则你的断言就没有任何意义。

2. 用assert代替if是陷阱之二。assert的判断和if语句差不多，但两者的作用有着本质的区别：assert关键字本意上是为测试调试程序时使用的，但如果不小心用assert来控制了程序的业务流程，那在测试调试结束后去掉assert关键字就意味着修改了程序的正常的逻辑。

3. assert断言失败将面临程序的退出。

因此，并不建议使用此关键字。相应的，在Hutool中封装了更加友好的Assert类，用于断言判定。

介绍

Assert类更像是Junit中的Assert类，也很像Guava中的Preconditions，主要作用是在方法或者任何地方对参数的有效性做校验。当不满足断言条件时，会抛出IllegalArgumentException或IllegalStateException异常。

使用

```
String a = null;
com.xiaoleilu.hutool.lang.Assert.isNull(a);
```

更方法

- isTrue 是否True
- isNull 是否是null值，不为null抛出异常
- notNull 是否非null值
- notEmpty 是否非空
- notBlank 是否非空白符
- notContain 是否为子串
- notEmpty 是否非空
- noNullElements 数组中是否包含null元素
- isInstanceOf 是否类实例
- isAssignable 是子类和父类关系
- state 会抛出IllegalStateException异常

JavaBean

概述

概述

针对JavaBean已经有BeanUtil的工具封装，我认为这还不够。最近看了Apache Commons BeanUtils的DynaBean源码和Nuts中Mirror类的文档（请看这里 -> 增强反射(http://nutzam.com/core/lang/mirror.html)），启发颇多，于是我决定在Hutool中加入DynaBean。

Dyna既Dynamic，顾名思义，通过Java反射机制操作JavaBean，以达到动态语言的某些特性。

bean包的另一个改进是针对PropertyDescriptor提供缓存。BeanInfoCache类缓存了通过内省获取到的PropertyDescriptor，以提高反射性能。

Bean工具-BeanUtil

什么是Bean

把一个拥有对属性进行set和get方法的类，我们就可以称之为JavaBean。实际上JavaBean就是一个Java类，在这个Java类中就默认形成了一种规则——对属性进行设置和获得。而反之将说Java类就是一个JavaBean，这种说法是错误的，因为一个java类中不一定有对属性的设置和获得的方法（也就是不一定有set和get方法）。

通常Java中对Bean的定义是包含setXXX和getXXX方法的对象，在Hutool中，采取一种简单的判定Bean的方法：是否存在只有一个参数的setXXX方法。

Bean工具类主要是针对这些setXXX和getXXX方法进行操作，比如将Bean对象转为Map等等。

方法

是否为Bean对象

BeanUtil.isBean方法根据是否存在只有一个参数的setXXX方法来判定是否是一个Bean对象。这样的判定方法主要目的是保证至少有一个setXXX方法用于属性注入。

```
boolean isBean = BeanUtil.isBean(HashMap.class);//false
```

内省 Introspector

把一类中需要进行设置和获得的属性访问权限设置为private（私有的）让外部的使用者看不见摸不着，而通过public（共有的）set和get方法来对其属性的值来进行设置和获得，而内部的操作具体是怎样的？外界使用的人不用不知道，这就称为内省。

Hutool中对内省的封装包括：

1. BeanUtil.getPropertyDescriptors 获得Bean字段描述数组
2. BeanUtil.getFieldNamePropertyDescriptorMap 获得字段名和字段描述Map
3. BeanUtil.getPropertyDescriptor 获得Bean类指定属性描述

Bean属性注入

BeanUtil.fillBean方法是bean注入的核心方法，此方法传入一个ValueProvider接口，通过实现此接口来获得key对应的值。CopyOptions参数则提供一些注入属性的选项。

CopyOptions的配置项包括：

1. editable 限制的类或接口，必须为目标对象的实现接口或父类，用于限制拷贝的属性，例如一个类我只想复制其父类的一些属性，就可以将editable设置为父类。
2. ignoreNullValue 是否忽略空值，当源对象的值为null时，true: 忽略而不注入此值，false: 注入null
3. ignoreProperties 忽略的属性列表，设置一个属性列表，不拷贝这些属性值
4. ignoreError 是否忽略字段注入错误

可以通过CopyOptions.create()方法创建一个默认的配置项，通过setXXX方法设置每个配置项。

ValueProvider接口需要实现两个方法：

1. value方法是通过key和目标类型来从任何地方获取一个值，并转换为目标类型，如果返回值不和目标类型匹配，将会自动调用Convert.convert方法转换。
2. containsKey方法主要是检测是否包含指定的key，如果不包含这个key，其对应的属性将会忽略注入。

首先定义一个bean：

```
public class Person{
private String name;
private int age;

public String getName() {
return name;
```

```
}
public void setName(String name) {
this.name = name;
}
public int getAge() {
return age;
}
public void setAge(int age) {
this.age = age;
}
}
```

然后注入这个bean：

```
Person person = BeanUtil.fillBean(new Person(), new ValueProvider<String>(){

@Override
public Object value(String key, Class<?> valueType) {
switch (key) {
case "name":
return "张三";
case "age":
return 18;
}
return null;
}

@Override
public boolean containsKey(String key) {
//总是存在key
return true;
}

}, CopyOptions.create());

Assert.assertEquals(person.getName(), "张三");
Assert.assertEquals(person.getAge(), 18);
```

同时，Hutool还提供了BeanUtil.toBean方法，此处并不是传Bean对象，而是Bean类，Hutool会自动调用默认构造方法创建对象。

基于BeanUtil.fillBean方法Hutool还提供了Map对象键值对注入Bean，其方法有：

1. BeanUtil.fillBeanWithMap
2. BeanUtil.fillBeanWithMapIgnoreCase

同时提供了map转bean的方法，与fillBean不同的是，此处并不是传Bean对象，而是Bean类，Hutool会自动调用默认构造方法创建对象。当然，前提是Bean类有默认构造方法（空构造），这些方法有：

1. BeanUtil.mapToBean
2. BeanUtil.mapToBeanIgnoreCase

在Java Web应用中，我们经常需要将ServletRequest对象中的参数注入bean（http表单数据），BeanUtil类提供了两个便捷方法：

1. BeanUtil.fillBeanWithRequestParam 将http表单数据注入Bean对象
2. BeanUtil.requestParamToBean 将http表单数据注入新建的Bean对象

Bean转为Map

BeanUtil.beanToMap方法则是将一个Bean对象转为Map对象。

Bean转Bean

Bean之间的转换主要是相同属性的复制，因此方法名为copyProperties。

BeanUtil.copyProperties方法同样提供一个CopyOptions参数用于自定义属性复制。

DynaBean

介绍

DynaBean是使用反射机制动态操作JavaBean的一个封装类，通过这个类，可以通过字符串传入name方式动态调用get和set方法，也可以动态创建JavaBean对象，亦或者执行JavaBean中的方法。

使用

我们先定义一个JavaBean：

```
public static class User{
private String name;
private int age;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public int getAge() {
return age;
```

```
}
public void setAge(int age) {
this.age = age;
}

public String testMethod(){
return "test for " + this.name;
}

@Override
public String toString() {
return "User [name=" + name + ", age=" + age + "]";
}
}
```

创建

```
DynaBean bean = DynaBean.create(user);
//我们也可以通过反射构造对象
DynaBean bean2 = DynaBean.create(User.class);
```

操作

我们通过DynaBean来包装并操作这个Bean

```
User user = new User();
DynaBean bean = DynaBean.create(user);
bean.set("name", "李华");
bean.set("age", 12);

String name = bean.get("name");//输出“李华”
```

这样我们就可以像操作Map一样动态操作JavaBean

invoke

除了标准的get和set方法，也可以调用invoke方法执行对象中的任意方法：

```
//执行指定方法
Object invoke = bean2.invoke("testMethod");
Assert.assertEquals("test for 李华", invoke);
```

“

说明:

DynaBean默认实现了hashCode、equals和toString三个方法，这三个方法也是默认调用原对象的相应方法操作。

表达式解析-BeanResolver

由来

很多JavaBean嵌套有很多层对象，这其中还夹杂着Map、Collection等对象，因此获取太深的嵌套对象会让代码变得冗长不堪。因此我们可以考虑使用一种表达式还获取指定深度的对象，于是BeanResolver应运而生。

原理

通过传入一个表达式，按照表达式的规则获取bean下指定的对象。

表达式分为两种：

- .表达式，可以获取Bean对象中的属性（字段）值或者Map中key对应的值
- []表达式，可以获取集合等对象中对应index的值

栗子：

1. persion 获取Bean对象下person字段的值，或者Bean本身如果是Person对象，返回本身。
2. persion.name 获取Bean中person字段下name字段的值，或者Bean本身如果是Person对象，返回其name字段的值。
3. persons[3] 获取persons字段下第三个元素的值（假设person是数组或Collection对象）
4. person.friends[5].name 获取person字段下friends列表（或数组）的第5个元素对象的name属性

使用

由于嵌套Bean定义过于复杂，在此我们省略，有兴趣的可以看下这里：com.xiaoleilu.hutool.core.lang.test.bean（src/test/java下）下定义了测试用例用的bean。

首先我们创建这个复杂的Bean（实际当中这个复杂的Bean可能是从数据库中获取，或者从JSON转入）

这个复杂Bean的关系是这样的：

定义一个Map包含用户信息（UserInfoDict）和一个标志位（flag），用户信息包括一些基本信息和一个考试信息列表（ExamInfoDict）。

```
//---------------------------------------------- 考试信息列表
ExamInfoDict examInfoDict = new ExamInfoDict();
examInfoDict.setId(1);
examInfoDict.setExamType(0);
examInfoDict.setAnswerIs(1);
```

```
ExamInfoDict examInfoDict1 = new ExamInfoDict();
examInfoDict1.setId(2);
examInfoDict1.setExamType(0);
examInfoDict1.setAnswerIs(0);

ExamInfoDict examInfoDict2 = new ExamInfoDict();
examInfoDict2.setId(3);
examInfoDict2.setExamType(1);
examInfoDict2.setAnswerIs(0);

List<ExamInfoDict> examInfoDicts = new ArrayList<ExamInfoDict>();
examInfoDicts.add(examInfoDict);
examInfoDicts.add(examInfoDict1);
examInfoDicts.add(examInfoDict2);

//---------------------------------------------- 用户信息
UserInfoDict userInfoDict = new UserInfoDict();
userInfoDict.setId(1);
userInfoDict.setPhotoPath("yx.mm.com");
userInfoDict.setRealName("张三");
userInfoDict.setExamInfoDict(examInfoDicts);

Map<String, Object> tempMap = new HashMap<String, Object>();
tempMap.put("userInfo", userInfoDict);
tempMap.put("flag", 1);
```

下面，我们使用BeanResolver获取这个Map下此用户第一门考试的ID：

```
BeanResolver resolver = new BeanResolver(tempMap, "userInfo.examInfoDict[0].id");
Object result = resolver.resolve();//ID为1
```

只需两句（甚至一句）即可完成复杂Bean中各层次对象的获取。

> 说明：
>
> 为了简化BeanResolver的使用，Hutool在BeanUtil中也加入了快捷入口方法：BeanUtil.getProperty，这个方法的命名更容易理解（毕竟BeanResolver不但可以解析Bean，而且可以解析Map和集合）。

集合类

概述

介绍

Iterator相关帮助类

集合包中封装了包括Enumeration、Iterator等的包装，这包括：

- ArrayIterator 数组Iterator，便于数组利用Iterator方式遍历
- CopiedIterator 为了解决并发情况下Iterator遍历导致的问题而封装的Iterator
- EnumerationIterator Enumeration的Iterator表现形式
- IteratorEnumeration Iterator的Enumeration表现形式

同时提供了IterUtil工具和CollUtil工具类用于简化对Iterator和集合的操作。

Map相关

为了便利，将部分Map的相关类归入collection包中。

- CaseInsensitiveMap Key大小写忽略的Map
- MapBuilder Map链式构建器
- MapProxy Map代理，通过代理方式代理一个Map，提供各种getXXX方法，并提供默认值支持

集合工具-CollUtil

集合工具 CollUtil

这个工具主要增加了对数组、集合类的操作。

1. join 方法

将集合转换为字符串，这个方法还是挺常用，是StrUtil.split的反方法。这个方法的参数支持各种类型对象的集合，最后连接每个对象时候调用其toString()方法。栗子如下：

```
String[] col= new String[]{"a","b","c","d","e"};
List<String> colList = CollUtil.newArrayList(col);

String str = CollUtil.join(colList, "#"); //str -> a#b#c#d#e
```

2. sortPageAll、sortPageAll2方法

这个方法其实是一个真正的组合方法，功能是：将给定的多个集合放到一个列表（List）中，根据给定的Comparator对象排序，然后分页取数据。这个方法非常类似于数据库多表查询后排序分页，这个方法存在的意义也是在此。sortPageAll2功能和sortPageAll的使用方式和结果是 一样的，区别是sortPageAll2使用了BoundedPriorityQueue这个类来存储组合后的列表，不知道哪种性能更好一些，所以就都保留了。使用此方法，栗子如下：

```
//Integer比较器
Comparator<Integer> comparator = new Comparator<Integer>(){
@Override
public int compare(Integer o1, Integer o2) {
return o1.compareTo(o2);
}
};

//新建三个列表，CollUtil.newArrayList方法表示新建ArrayList并填充元素
List<Integer> list1 = CollUtil.newArrayList(1, 2, 3);
List<Integer> list2 = CollUtil.newArrayList(4, 5, 6);
List<Integer> list3 = CollUtil.newArrayList(7, 8, 9);

//参数表示把list1,list2,list3合并并按照从小到大排序后，取0~2个（包括第0个，不包括第2个），结果是[1,2]
@SuppressWarnings("unchecked")
List<Integer> result = CollUtil.sortPageAll(0, 2, comparator, list1, list2, list3);
System.out.println(result);     //输出 [1,2]
```

3. sortEntrySetToList方法

这个方法主要是对Entry<Long, Long>按照Value的值做排序，使用局限性较大，我已经忘记哪里用到过了……

4. popPart方法

这个方法传入一个栈对象，然后弹出指定数目的元素对象，弹出是指pop()方法，会从原栈中删掉。

5.newHashMap、newHashSet、newArrayList方法

这些方法是新建相应的数据结构，数据结构元素的类型取决于你变量的类型，栗子如下：

```
HashMap<String, String> map = CollUtil.newHashMap();
HashSet<String> set = CollUtil.newHashSet();
ArrayList<String> list = CollUtil.newArrayList();
```

6. append方法

在给定数组里末尾加一个元素，其实List.add()也是这么实现的，这个方法存在的意义是只有少量的添加元素时使用，因为内部使用了System.arraycopy,每调用一次就要拷贝数组一次。这个方法也是为了在某些只能使用数组的情况下使用，省去了先要转成List，添加元素，再转成Array。

7. resize方法

重新调整数据的大小，如果调整后的大小比原来小，截断，如果比原来大，则多出的位置空着。（貌似List在扩充的时候会用到类似的方法）

8. addAll方法

将多个数据合并成一个数组

9. range方法

这个方法来源于Python(https://www.python.org/)的一个语法糖，给定开始和结尾以及步进，就会生成一个等差数列（列表）

```
int[] a1 = CollUtil.range(6);       //[0,1,2,3,4,5]
int[] a2 = CollUtil.range(4, 7);    //[4,5,6]
int[] a3 = CollUtil.range(4, 9, 2); //[4,6,8]
````

10. `sub`方法
对集合切片，其他类型的集合会转换成`List`，封装`List.subList`方法，自动修正越界等问题，完全避免`IndexOutOfBoundsException`异常。

11. `isEmpty`、`isNotEmpty`方法
判断集合是否为空（包括null和没有元素的集合）。

12. `zip`方法
此方法也是来源于[Python](https://www.python.org/)的一个语法糖，给定两个集合，然后两个集合中的元素一一对应，成为一个Map。此方法还有一个重载方法，可以传字符，然后给定分分隔符，字符串会被split成列表。栗子：
```Java
String[] keys = new String[]{"a", "b", "c"};
Integer[] values = new Integer[]{1, 2, 3};
Map<String, Integer> map = CollUtil.zip(keys,values);
System.out.println(map);    // {b=2, c=3, a=1}

String a = "a,b,c";
String b = "1,2,3";
Map<String, String> map2 = CollUtil.zip(a,b, ",");
System.out.println(map2);   // {b=2, c=3, a=1}
```

13. filter方法

此方法可以过滤map，排除不需要的key。栗子：

```
@Test
public void CollUtil_Filter() {
    Map<String, Object> m = new HashMap<String, Object>() {{
        put("k1", "v1");
        put("k2", "v2");
        put("k3", "v3");
    }};
    String[] inc = {"k1", "k3"};//需要的key
    List<String> incList = Arrays.asList(inc);
    m = CollectionUtil.filter(m, new Editor<Map.Entry<String, Object>>() {
        @Override
        public Map.Entry<String, Object> edit(Map.Entry<String, Object>
stringObjectEntry) {
            if (incList.contains(stringObjectEntry.getKey())) {
                return stringObjectEntry;
            }
            return null;
        }
    });
    log.info("{}", m);
}
```

结果

```
{k3=v3, k1=v1}
```

Iterator工具-IterUtil

来源

最早此工具类中的方法是在CollUtil中的，由于经过抽象，因此单独拿出来以适应更广的场景。

方法介绍

- isEmpty 是否为null或者无元素
- isNotEmpty 是否为非null或者至少一个元素
- hasNull 是否有null元素
- isAllNull 是否全部为null元素
- countMap 根据集合返回一个元素计数的Map，所谓元素计数就是假如这个集合中某个元素出现了n次，那将这个元素做为key，n做为value
- join 使用分隔符将集合转换为字符串

- toMap toMap Entry列表转Map，或者key和value单独列表转Map
- asIterator Enumeration转Iterator
- asIterable Iterator转Iterable
- getFirst 获取列表的第一个元素
- getElementType 获取元素类型

Map工具

概述

由来

最早Map的相关工具是被包含在CollUtil中的，但是考虑到Map和集合非同一类数据结构，因此独立出来，且Hutool封装了许多自定义的Map类，因此单独成包。

使用

特殊Map

- CaseInsensitiveMap 忽略大小写的Map,对KEY忽略大小写，get("Value")和get("value")获得的值相同，put进入的值也会被覆盖
- CaseInsensitiveLinkedMap 忽略大小写的LinkedHashMap,对KEY忽略大小写，get("Value")和get("value")获得的值相同，put进入的值也会被覆盖
- MapBuilder Map创建器，可以链式创建Map
- MapProxy Map代理类，通过代理包装Map，提供一系列的getXXX方法

工具

- MapUtil 提供对Map常用操作的封装

Map工具-MapUtil

介绍

MapUtil是针对Map的一一列工具方法的封装，包括getXXX的快捷值转换方法。

方法

- isEmpty、isNotEmpty 判断Map为空和非空方法，空的定义为null或没有值
- newHashMap 快速创建多种类型的HashMap实例

• createMap 创建自定义的Map类型的Map

• of 此方法将一个或多个键值对加入到一个新建的Map中，下面是栗子:

```
Map<Object, Object> colorMap = MapUtil.of(new String[][] {{
    {"RED", "#FF0000"},
    {"GREEN", "#00FF00"},
    {"BLUE", "#0000FF"}
});
```

• toListMap 行转列，合并相同的键，值合并为列表，将Map列表中相同key的值组成列表做为Map的value，例如传入数据是：

```
[
  {a: 1, b: 1, c: 1}
  {a: 2, b: 2}
  {a: 3, b: 3}
  {a: 4}
]
```

结果为：

```
{
   a: [1,2,3,4]
   b: [1,2,3,]
   c: [1]
}
```

• toMapList 列转行。将Map中值列表分别按照其位置与key组成新的map，例如传入数据：

```
{
   a: [1,2,3,4]
   b: [1,2,3,]
   c: [1]
}
```

结果为：

```
[
  {a: 1, b: 1, c: 1}
  {a: 2, b: 2}
  {a: 3, b: 3}
  {a: 4}
]
```

• join、joinIgnoreNull 将Map按照给定的分隔符转换为字符串

• filter 过滤过程通过传入的Editor实现来返回需要的元素内容，这个Editor实现可以实现以下功能

：1、过滤出需要的对象，如果返回null表示这个元素对象抛弃 2、修改元素对象，返回集合中为修改后的对象

- reverse Map的键和值互换
- sort 排序Map
- getAny 获取Map的部分key生成新的Map
- get、getXXX 获取Map中指定类型的值

Codec编码

Base64编码解码-Base64

介绍

Base64编码是用64（2的6次方）个ASCII字符来表示256（2的8次方）个ASCII字符，也就是三位二进制数组经过编码后变为四位的ASCII字符显示，长度比原来增加1/3。

使用

```
String a = "伦家是一个非常长的字符串";
String encode = Base64.encode(a);
Assert.assertEquals("5Lym5a625piv5LiA5Liq6Z2e5bi46ZW/55qE5a2X56ym5Liy",
encode);

String decodeStr = Base64.decodeStr(encode);
Assert.assertEquals(a, decodeStr);
```

Base32编码解码-Base32

介绍

Base32就是用32（2的5次方）个特定ASCII码来表示256个ASCII码。所以，5个ASCII字符经过base32编码后会变为8个字符（公约数为40），长度增加3/5.不足8n用“=”补足。

使用

```
String a = "伦家是一个非常长的字符串";

String encode = Base32.encode(a);
Assert.assertEquals("4S6KNZNOW3TJRL7EXCAOJOFK5GOZ5ZNYXDUZLP7HTKCOLLMX
46WKNZFYWI", encode);
```

```
String decodeStr = Base32.decodeStr(encode);
Assert.assertEquals(a, decodeStr);
```

文本操作

CSV文件处理工具-CsvUtil

介绍

逗号分隔值（Comma-Separated Values，CSV，有时也称为字符分隔值，因为分隔字符也可以不是逗号），其文件以纯文本形式存储表格数据（数字和文本）。

Hutool针对此格式，参考FastCSV项目做了对CSV文件读写的实现(Hutool实现完全独立，不依赖第三方)

CsvUtil是CSV工具类，主要封装了两个方法：

- getReader 用于对CSV文件读取
- getWriter 用于生成CSV文件

这两个方法分别获取CsvReader对象和CsvWriter，从而独立完成CSV文件的读写。

使用

读取CSV文件

```
CsvReader reader = CsvUtil.getReader();
//从文件中读取CSV数据
CsvData data = reader.read(FileUtil.file("test.csv"));
List<CsvRow> rows = data.getRows();
//遍历行
for (CsvRow csvRow : data) {
//getRawList返回一个List列表，列表的每一项为CSV中的一个单元格（既逗号分隔部分）
Console.log(csvRow.getRawList());
}
```

CsvRow对象还记录了一些其他信息，包括原始行号等。

生成CSV文件

```
//指定路径和编码
CsvWriter writer = CsvUtil.getWriter("e:/testWrite.csv", CharsetUtil.CHARSET_UTF_8);
```

```
//按行写出
writer.write(
new String[] {"a1", "b1", "c1"},
new String[] {"a2", "b2", "c2"},
new String[] {"a3", "b3", "c3"}
);
```

效果如下：

图片地址：https://static.oschina.net/uploads/img/201809/05222906_kF5o.png

> 注意
>
> CSV文件本身为一种简单文本格式，有编码区分。Excel读取CSV文件中含有中文时时必须为GBK编码（Windows平台下），否则会出现乱码。

Unicode编码转换工具-UnicodeUtil

介绍

此工具主要针对类似于\u4e2d\u6587这类Unicode字符做一些特殊转换。

使用

字符串转Unicode符

```
//第二个参数true表示跳过ASCII字符（只跳过可见字符）
String s = UnicodeUtil.toUnicode("aaa123中文", true);
//结果aaa123\\u4e2d\\u6587
```

Unicode转字符串

```
String str = "aaa\\U4e2d\\u6587\\u111\\urtyu\\u0026";
String res = UnicodeUtil.toString(str);
//结果aaa中文\\u111\\urtyu&
```

由于\u111为非Unicode字符串，因此原样输出。

可复用字符串生成器-StrBuilder

介绍

在JDK提供的StringBuilder中，拼接字符串变得更加高效和灵活，但是生成新的字符串需要重新构

建StringBuilder对象，造成性能损耗和内存浪费，因此Hutool提供了可复用的StrBuilder。

使用

StrBuilder和StringBuilder使用方法基本一致，只是多了reset方法可以重新构建一个新的字符串而不必开辟新内存。

```
StrBuilder builder = StrBuilder.create();
builder.append("aaa").append("你好").append('r');
//结果：aaa你好r
```

多次构建字符串性能测试

我们模拟创建1000000次字符串对两者性能对比，采用TimeInterval计时：

```
//StringBuilder
TimeInterval timer = DateUtil.timer();
StringBuilder b2 = new StringBuilder();
for(int i =0; i< 1000000; i++) {
b2.append("test");
b2 = new StringBuilder();
}
Console.log(timer.interval());
```

```
//StrBuilder
TimeInterval timer = DateUtil.timer();
StrBuilder builder = StrBuilder.create();
for(int i =0; i< 1000000; i++) {
builder.append("test");
builder.reset();
}
Console.log(timer.interval());
```

测试结果为：

```
StringBuilder: 39ms
StrBuilder   : 20ms
```

性能几乎翻倍。也欢迎用户自行测试。

注解

注解工具-AnnotationUtil

介绍

封装了注解获取等方法的工具类。

使用

方法介绍

1. 注解获取相关方法：

• getAnnotations 获取指定类、方法、字段、构造等上的注解列表

• getAnnotation 获取指定类型注解

• getAnnotationValue 获取指定注解属性的值

1. 注解属性获取相关方法：

• getRetentionPolicy 获取注解类的保留时间，可选值 SOURCE（源码时），CLASS（编译时），RUNTIME（运行时），默认为 CLASS

• getTargetType 获取注解类可以用来修饰哪些程序元素，如 TYPE, METHOD, CONSTRUCTOR, FIELD, PARAMETER 等

• isDocumented 是否会保存到 Javadoc 文档中

• isInherited 是否可以被继承，默认为 false

更多方法见API文档：

https://apidoc.gitee.com/loolly/hutool/cn/hutool/core/annotation/AnnotationUtil.html(https://apidoc.gitee.com/loolly/hutool/cn/hutool/core/annotation/AnnotationUtil.html)

比较器

介绍

各种比较器（Comparator）实现和封装

提供的比较器

• ReverseComparator 反转比较器，排序时提供反序

• VersionComparator 版本比较器，支持如：1.3.20.8，6.82.20160101，8.5a/8.5c等版本形式

• PropertyComparator Bean属性比较器，通过Bean的某个属性来对Bean对象进行排序

• IndexedComparator 按照数组的顺序正序排列，数组的元素位置决定了对象的排序先后

• ComparatorChain 比较器链。此链包装了多个比较器，最终比较结果按照比较器顺序综合多个比较器结果。

• PinyinComparator 按照GBK拼音顺序对给定的汉字字符串排序。

异常

异常工具-ExceptionUtil

介绍

针对异常封装，例如包装为RuntimeException。

方法

- getMessage 获得完整消息，包括异常名
- wrap 包装一个异常为指定类型异常
- wrapRuntime 使用运行时异常包装编译异常
- getCausedBy 获取由指定异常类引起的异常
- isCausedBy 判断是否由指定异常类引起
- stacktraceToString 堆栈转为完整字符串

其它方法见API文档：

https://apidoc.gitee.com/loolly/hutool/cn/hutool/core/exceptions/ExceptionUtil.html(https://apidoc.gitee.com/loolly/hutool/cn/hutool/core/exceptions/ExceptionUtil.html)

其它异常封装

介绍

针对Hutool中常见异常封装

异常类

- DependencyException 依赖异常
- StatefulException 带有状态码的异常
- UtilException 工具类异常
- NotInitedException 未初始化异常
- ValidateException 验证异常

数学

数学相关-MathUtil

介绍

此工具是NumberUtil的一个补充，NumberUtil偏向于简单数学计算的封装，MathUtil偏向复杂数学计算。

方法

1. 排列

- arrangementCount 计算排列数
- arrangementSelect 排列选择（从列表中选择n个排列）

1. 组合

- combinationCount 计算组合数，即C(n, m) = n!/((n-m)! * m!)
- combinationSelect 组合选择（从列表中选择n个组合）

线程和并发

线程工具-ThreadUtil

由来

并发在Java中算是一个比较难理解和容易出问题的部分，而并发的核心在线程。好在从JDK1.5开始Java提供了concurrent包可以很好的帮我们处理大部分并发、异步等问题。

不过，ExecutorService和Executors等众多概念依旧让我们使用这个包变得比较麻烦，如何才能隐藏这些概念？又如何用一个方法解决问题？ThreadUtil便为此而生。

原理

Hutool使用GlobalThreadPool持有一个全局的线程池，默认所有异步方法在这个线程池中执行。

方法

ThreadUtil.execute

直接在公共线程池中执行线程

ThreadUtil.newExecutor

获得一个新的线程池

ThreadUtil.excAsync

执行异步方法

ThreadUtil.newCompletionService

创建CompletionService，调用其submit方法可以异步执行多个任务，最后调用take方法按照完成的顺序获得其结果。若未完成，则会阻塞。

ThreadUtil.newCountDownLatch

新建一个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。

ThreadUtil.sleep

挂起当前线程，是Thread.sleep的封装，通过返回boolean值表示是否被打断，而不是抛出异常。

> ThreadUtil.safeSleep方法是一个保证挂起足够时间的方法，当给定一个挂起时间，使用此方法可以保证挂起的时间大于或等于给定时间，解决Thread.sleep挂起时间不足问题，此方法在Hutool-cron的定时器中使用保证定时任务执行的准确性。

ThreadUtil.getStackTrace

此部分包括两个方法：

- getStackTrace 获得堆栈列表
- getStackTraceElement 获得堆栈项

其它

- createThreadLocal 创建本地线程对象
- interupt 结束线程，调用此方法后，线程将抛出InterruptedException异常
- waitForDie 等待线程结束. 调用 Thread.join() 并忽略 InterruptedException
- getThreads 获取JVM中与当前线程同组的所有线程
- getMainThread 获取进程的主线程

配置文件(Hutool-setting)

配置文件模块概述

由来

Setting

众所周知，Java中广泛应用的配置文件Properties存在一个特别大的诟病：不支持中文。每次使用时，如果想存放中文字符，必须借助IDE相关插件才能转为Unicode符号，而这种反人类的符号在命令行下根本没法看（想想部署在服务器上后修改配置文件是一件多么痛苦的事情）

于是，在很多框架中开始渐渐抛弃Properties文件而转向XML配置文件（例如Hibernate和Spring早期版本）。但是XML罗嗦的配置方式实在无法忍受。于是，Setting诞生。

Props

Properties的第二个问题是读取非常不方便，需要我们自己写长长的代码进行load操作：

```
properties = new Properties();
try {
Class clazz = Demo1.class;
InputStream inputestream = clazz.getResourceAsStream("db.properties");
properties.load( inputestream);
}catch (IOException e) {
//ignore
}
```

而Props则大大简化为：

```
Props props = new Props("db.properties");
```

考虑到Properties使用依旧广泛，因此封装了Props类以应对兼容性。

设置文件-Setting

简介

Setting除了兼容Properties文件格式外，还提供了一些特有功能，这些功能包括：

- 各种编码方式支持
- 变量支持
- 分组支持

首先说编码支持，在Properties中，只支ISO8859-1导致在Properties文件中注释和value没法使用中文，（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），因此Setting中引入自定义编码，可以很好的支持各种编码的配置文件。

再就是变量支持，在Setting中，支持${key}变量，可以将之前定义的键对应的值做为本条值得一部分，这个特性可以减少大量的配置文件冗余。

最后是分组支持。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，具体大家可以自行百度之。当然，在Windows的ini文件中也有分组的概念，Setting将这一概念引入，从而大大增加配置文件的可读性。

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${demo.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。同时在value之后不允许有注释，会被当作value的一部分。

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("\\$\\{(.*?)\\}");
}
}
```

Properties扩展-Props

介绍

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。

使用

Props的使用方法和Properties以及Setting一致（同时支持）：

```
Props props = new Props("test.properties");
String user = props.getProperty("user");
String driver = props.getStr("driver");
```

日志(Hutool-log)

Log模块概述

由来

准确的说，Hutool-log只是一个日志的通用门面，功能类似于Slf4j。既然像Slf4j这种门面框架已经

非常完善，为何还要自己做一个门面呢？下面我列举实践中遇到的一些问题：

已有门面存在问题

1. log对象创建比较复杂

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 final Logger log = LoggerFactory.getLogger(XXX.class);
```

1. 对于附带Exception参数的方法，并不支持变量。

Slf4j中我最喜欢的形式，这样既省去了麻烦的isInfoEnabled()的判断，还避免了拼接字符串：

```
log.info("我在XXX 改了 {} 变量", "name");
```

但是这种情况下就无法使用变量模式：

```
log.error("错误消息", e);
```

特点

1. Logfactory.get方法不再需要（或者不是必须）传入当前类名，会自定解析当前类名
2. log.xxx方法在传入Exception时同时支持模板语法。
3. 不需要桥接包而自动适配引入的日志框架，在无日志框架下依旧可以完美适配JDK Logging。
4. 引入多个日志框架情况下，可以自定义日志框架输出。

原理

Hutool-log采用动态自动适配模式，它会自动检测引入的日志框架包从而将日志输出到此框架。

比如我们在项目中引入Log4j的包，Hutool-log会自动检测到此包的存在，并将日志输出到log4j。如果没有引入任何日志框架，会将日志输出到JDK Logging中。

因此，Hutool-log并没有统一的配置文件，如果你引入任何一种日志框架，使用此框架的配置文件即可。

Hutool-log对于日志框架的监测顺序是：

Slf4j(Logback) > Log4j > Log4j2 > Apache Commons Logging > JDK Logging > Console

当然，如果只是引入Slf4j-API，而没有引入任何实现，Slf4j将被跳过。

> 关于日志框架监测的核心代码请参阅：LogFactory.create

使用

常规使用

Hutool-log的使用与一般日志框架并无区别，调用LogFactory.get()即可简单获取Log实现对象。

```
Log log = LogFactory.get();

log.debug("This is {} log", Level.DEBUG);
log.info("This is {} log", Level.INFO);
log.warn("This is {} log", Level.WARN);

Exception e = new Exception("test Exception");
log.error(e, "This is {} log", Level.ERROR);
```

> 通常我们需要在类中定义日志为private static final Log log = LogFactory.get();以获得更好的性能。

自定义日志实现

有的时候，我们需要自定义日志框架输出，这是我们就需要调用LogFactory.setCurrentLogFactory方法来定义全局的日志实现。

```
// 自动选择日志实现
Log log = LogFactory.get();
log.debug("This is {} log", "default");
Console.log("---------------------------------------------------------------");

//自定义日志实现为Apache Commons Logging
LogFactory.setCurrentLogFactory(new ApacheCommonsLogFactory());
log.debug("This is {} log", "custom apache commons logging");
Console.log("---------------------------------------------------------------");

//自定义日志实现为JDK Logging
LogFactory.setCurrentLogFactory(new JdkLogFactory());
log.info("This is {} log", "custom jdk logging");
Console.log("---------------------------------------------------------------");

//自定义日志实现为Console Logging
LogFactory.setCurrentLogFactory(new ConsoleLogFactory());
log.info("This is {} log", "custom Console");
Console.log("---------------------------------------------------------------");
```

> 默认的，在未发现任何第三方日志的情况下，检查logging.properties文件是否在classpath中存在（鉴定用户是否想用JDK Logging），如果没有这个配置文件，默认是按照Hutool预定义规则打印到控制台。

日志工厂-LogFactory

介绍

Hutool-log做为一个日志门面，为了兼容各大日志框架，一个用于自动创建日志对象的日志工厂类必不可少。

LogFactory类用于灵活的创建日志对象，通过static方法创建我们需要的日志，主要功能如下：

- LogFactory.get 自动识别引入的日志框架，从而创建对应日志框架的门面Log对象（此方法创建一次后，下次再次get会根据传入类名缓存Log对象，对于每个类，Log对象都是单例的），同时自动识别当前类，将当前类做为类名传入日志框架。
- LogFactory.createLog 与get方法作用类似。但是此方法调用后会每次创建一个新的Log对象。
- LogFactory.setCurrentLogFactory 自定义当前日志门面的日志实现类。当引入多个日志框架时，我们希望自定义所用的日志框架，调用此方法即可。需要注意的是，此方法为全局方法，在获取Log对象前只调用一次即可。

使用

获取当前类对应的Log对象：

```
//推荐创建不可变静态类成员变量
private static final Log log = LogFactory.get();
```

如果你想获得自定义name的Log对象（像普通Log日志实现一样），那么可以使用如下方式获取Log：

```
private static final Log log = LogFactory.get("我是一个自定义日志名");
```

自定义日志实现

```
//自定义日志实现为Apache Commons Logging
LogFactory.setCurrentLogFactory(new ApacheCommonsLogFactory());

//自定义日志实现为JDK Logging
LogFactory.setCurrentLogFactory(new JdkLogFactory());

//自定义日志实现为Console Logging
LogFactory.setCurrentLogFactory(new ConsoleLogFactory());
```

自定义日志工厂（自定义日志门面实现）

LogFactory是一个抽象类，我们可以继承此类，实现createLog方法即可（同时我们可能需要实现Log接口来达到自定义门面的目的），这样我们就可以自定义一个日志门面。最后通过LogFactory.setCurrentLogFactory方法装入这个自定义LogFactory即可实现自定义日志门面。

“

PS

自定义日志门面的实现可以参考cn.hutool.log.dialect包中的实现内容自定义扩展。

本质上，实现Log接口，做一个日志实现的Wrapper，然后在相应的工厂类中创建此Log实例即可。同时，LogFactory中还可以初始化一些启动配置参数。

静态调用日志-StaticLog

由来

很多时候，我们只是想简简单的使用日志，最好一个方法搞定，我也不想创建Log对象，那么StaticLog或许是你需要的。

使用

```
StaticLog.info("This is static {} log.", "INFO");
```

同样StaticLog提供了trace、debug、info、warn、error方法，提供变量占位符支持，使项目中日志的使用简单到没朋友。

StaticLog类中同样提供log方法，可能在极致简洁的状况下，提供非常棒的灵活性（打印日志等级由level参数决定）

与LogFactory同名方法

假如你只知道StaticLog，不知道LogFactory怎么办？Hutool非常贴心的提供了get方法，此方法与Logfactory中的get方法一样，同样可以获得Log对象。

疑惑解答

问：

程序中出现[WARN] Can not find [logging.properties], use [%JRE_HOME%/lib/logging.properties] as default!这行警告，请问这个问题如何解决？

答：

这是因为你没有引入任何第三方日志框架。Hutool默认会使用JDK Logging做为其日志实现。出现这句话的意思是你没有在你的ClassPath下放logging.properties（JDK Logging的配置文件），系统默认默认读取JDK目录下的默认配置文件。你可以考虑在ClassPath下放一个logging.properties（就是src/main/resources）下。

配置文件模板见：https://gitee.com/loolly/hutool/blob/v4-master/hutool-log/src/test/resources/logging.properties(https://gitee.com/loolly/hutool/blob/v4-master/hutool-log/src/test/resources/logging.properties)

其它日志框架的配置文件模板见：https://gitee.com/loolly/hutool/tree/v4-master/hutool-log/src/test/resources(https://gitee.com/loolly/hutool/tree/v4-master/hutool-log/src/test/resources)

> 注意
>
> 在新版的Hutool中会自动检测logging.properties存在与否，如果不存在这个配置文件，将跳过JDK-Logging从而使用Console-log（命令行打印日志）

缓存(Hutool-cache)

概述

来源

Hutool-cache模块最早受到jodd-cache的启发（如今大部分逻辑依旧与jodd保持一致），此模块提供一种缓存的简单实现方案，在小型项目中对于简单的缓存需求非常好用。

介绍

Hutoo-cache模块提供了几种缓存策略实现：

FIFOCache

FIFO(first in first out) 先进先出策略。元素不停的加入缓存直到缓存满为止，当缓存满时，清理过期缓存对象，清理后依旧满则删除先入的缓存（链表首部对象）。

优点：简单快速

缺点：不灵活，不能保证最常用的对象总是被保留

LFUCache

LFU(least frequently used) 最少使用率策略。根据使用次数来判定对象是否被持续缓存（使用率是通过访问次数计算），当缓存满时清理过期对象，清理后依旧满的情况下清除最少访问（访问计数最小）的对象并将其他对象的访问数减去这个最小访问数，以便新对象进入后可以公平计数。

LRUCache

LRU (least recently used)最近最久未使用缓存。根据使用时间来判定对象是否被持续缓存，当对象被访问时放入缓存，当缓存满了，最久未被使用的对象将被移除。此缓存基于LinkedHashMap，因此当被缓存的对象每被访问一次，这个对象的key就到链表头部。这个算法简单并且非常快，他比FIFO有一个显著优势是经常使用的对象不太可能被移除缓存。缺点是当缓存满时，不能被很快的访问。

TimedCache

定时缓存，对被缓存的对象定义一个过期时间，当对象超过过期时间会被清理。此缓存没有容量限制，对象只有在过期后才会被移除

WeakCache

弱引用缓存。对于一个给定的键，其映射的存在并不阻止垃圾回收器对该键的丢弃，这就使该键成为可终止的，被终止，然后被回收。丢弃某个键时，其条目从映射中有效地移除。该类使用了WeakHashMap做为其实现，缓存的清理依赖于JVM的垃圾回收。

FileCach

FileCach是一个独立的缓存，主要是将小文件以byte[]的形式缓存到内容中，减少文件的访问，以解决频繁读取文件引起的性能问题。

主要实现有：

- LFUFileCache
- LRUFileCache

CacheUtil

概述

CacheUtil是缓存创建的快捷工具类。用于快速创建不同的缓存对象。

使用

```
//新建FIFOCache
Cache<String,String> fifoCache = CacheUtil.newFIFOCache(3);
```

同样其它类型的Cache也可以调用newXXX的方法创建。

FIFOCache

介绍

FIFO(first in first out) 先进先出策略。元素不停的加入缓存直到缓存满为止，当缓存满时，清理过期缓存对象，清理后依旧满则删除先入的缓存（链表首部对象）。

优点：简单快速

缺点：不灵活，不能保证最常用的对象总是被保留

使用

```
Cache<String,String> fifoCache = CacheUtil.newFIFOCache(3);

//加入元素，每个元素可以设置其过期时长，DateUnit.SECOND.getMillis()代表每秒对应的毫秒数，在此为3秒
fifoCache.put("key1", "value1", DateUnit.SECOND.getMillis() * 3);
fifoCache.put("key2", "value2", DateUnit.SECOND.getMillis() * 3);
fifoCache.put("key3", "value3", DateUnit.SECOND.getMillis() * 3);

//由于缓存容量只有3，当加入第四个元素的时候，根据FIFO规则，最先放入的对象将被移除
fifoCache.put("key4", "value4", DateUnit.SECOND.getMillis() * 3);

//value1为null
String value1 = fifoCache.get("key1");
```

LFUCache

介绍

LFU(least frequently used) 最少使用率策略。根据使用次数来判定对象是否被持续缓存（使用率是通过访问次数计算），当缓存满时清理过期对象，清理后依旧满的情况下清除最少访问（访问计数最小）的对象并将其他对象的访问数减去这个最小访问数，以便新对象进入后可以公平计数。

使用

```
Cache<String, String> lfuCache = CacheUtil.newLFUCache(3);
//通过实例化对象创建
//LFUCache<String, String> lfuCache = new LFUCache<String, String>(3);

lfuCache.put("key1", "value1", DateUnit.SECOND.getMillis() * 3);
lfuCache.get("key1");//使用次数+1
lfuCache.put("key2", "value2", DateUnit.SECOND.getMillis() * 3);
lfuCache.put("key3", "value3", DateUnit.SECOND.getMillis() * 3);
lfuCache.put("key4", "value4", DateUnit.SECOND.getMillis() * 3);

//由于缓存容量只有3，当加入第四个元素的时候，根据LRU规则，最少使用的将被移除（2,3被移除）
String value2 = lfuCache.get("key2");//null
String value3 = lfuCache.get("key3");//null
```

LRUCache

介绍

LRU (least recently used)最近最久未使用缓存。根据使用时间来判定对象是否被持续缓存，当对象被访问时放入缓存，当缓存满了，最久未被使用的对象将被移除。此缓存基于LinkedHashMap，因此当被缓存的对象每被访问一次，这个对象的key就到链表头部。这个算法简单并且非常快，他比FIFO有一个显著优势是经常使用的对象不太可能被移除缓存。缺点是当缓存满时，不能被很快的访问。

使用

```
Cache<String, String> lruCache = CacheUtil.newLRUCache(3);
//通过实例化对象创建
//LRUCache<String, String> lruCache = new LRUCache<String, String>(3);
lruCache.put("key1", "value1", DateUnit.SECOND.getMillis() * 3);
lruCache.put("key2", "value2", DateUnit.SECOND.getMillis() * 3);
lruCache.put("key3", "value3", DateUnit.SECOND.getMillis() * 3);
lruCache.get("key1");//使用时间推近
lruCache.put("key4", "value4", DateUnit.SECOND.getMillis() * 3);

//由于缓存容量只有3，当加入第四个元素的时候，根据LRU规则，最少使用的将被移除（2被移除）
String value2 = lruCache.get("key");//null
```

TimedCache

介绍

定时缓存，对被缓存的对象定义一个过期时间，当对象超过过期时间会被清理。此缓存没有容量限制，对象只有在过期后才会被移除。

使用

```
//创建缓存，默认4毫秒过期
TimedCache<String, String> timedCache = CacheUtil.newTimedCache(4);
//实例化创建
//TimedCache<String, String> timedCache = new TimedCache<String, String>(4);

timedCache.put("key1", "value1", 1);//1毫秒过期
timedCache.put("key2", "value2", DateUnit.SECOND.getMillis() * 5);
```

```
timedCache.put("key3", "value3");//默认过期(4毫秒)

//启动定时任务，每5毫秒秒检查一次过期
timedCache.schedulePrune(5);

//等待5毫秒
ThreadUtil.sleep(5);

//5毫秒后由于value2设置了5毫秒过期，因此只有value2被保留下来
String value1 = timedCache.get("key1");//null
String value2 = timedCache.get("key2");//value2

//5毫秒后，由于设置了默认过期，key3只被保留4毫秒，因此为null
String value3 = timedCache.get("key3");//null

//取消定时清理
timedCache.cancelPruneSchedule();
```

如果用户在超时前调用了get(key)方法，会重头计算起始时间。举个例子，用户设置key1的超时时间5s，用户在4s的时候调用了get("key1")，此时超时时间重新计算，再过4s调用get("key1")方法值依旧存在。如果想避开这个机制，请调用get("key1", false)方法。

> 说明
>
> 如果启动了定时器，那会定时清理缓存中的过期值，但是如果不起动，那只有在get这个值得时候才检查过期并清理。不起动定时器带来的问题是：有些值如果长时间不访问，会占用缓存的空间。

WeakCache

介绍

弱引用缓存。对于一个给定的键，其映射的存在并不阻止垃圾回收器对该键的丢弃，这就使该键成为可终止的，被终止，然后被回收。丢弃某个键时，其条目从映射中有效地移除。该类使用了WeakHashMap做为其实现，缓存的清理依赖于JVM的垃圾回收。

使用

与TimedCache使用方法一致：

```
WeakCache<String, String> weakCache =
CacheUtil.newWeakCache(DateUnit.SECOND.getMillis() * 3);
```

WeakCache也可以像TimedCache一样设置定时清理时间，同时具备垃圾回收清理。

FileCache

介绍

FileCache主要是将小文件以byte[]的形式缓存到内容中，减少文件的访问，以解决频繁读取文件引起的性能问题。

实现

- LFUFileCache
- LRUFileCache

使用

```
//参数1：容量，能容纳的byte数
//参数2：最大文件大小，byte数，决定能缓存至少多少文件，大于这个值不被缓存直接读取
//参数3：超时。毫秒
LFUFileCache cache = new LFUFileCache(1000, 500, 2000);
byte[] bytes = cache.getFileBytes("d:/a.jpg");
```

LRUFileCache的使用与LFUFileCache一致，不再举例。

JSON(Hutool-json)

概述

Hutool-json

为何集成

JSON在现在的开发中做为跨平台的数据交换格式已经慢慢有替代XML的趋势（比如RestFul规范），我想大家在开发中对外提供接口也越来越多的使用JSON格式。

不可否认，现在优秀的JSON框架非常多，我经常使用的像阿里的FastJSON，Jackson等都是非常优秀的包，性能突出，简单易用。Hutool开始也并不想自己写一个JSON，但是在各种工具的封装中，发现JSON已经不可或缺，因此将json.org官方的JSON解析纳入其中，进行改造。在改造过程中，积极吸取其它类库优点，优化成员方法，抽象接口和类，最终形成Hutool-json。

介绍

Hutool-json的核心类只有两个：

• JSONObject

• JSONArray

这与其它JSON包是类似的，与此同时，还提供一个JSONUtil工具类用于简化针对JSON的各种操作和转换。

除了核心类，还提供了一些辅助类用于实现特定功能：

• JSONSupport Bean类继承此对象即可无缝转换为JSON或JSON字符串。同时实现了toString()方法可将当前对象输出为JSON字符串。

• XML 提供JSON与XML之间的快速转换，同时JSONUtil中有相应静态封装。

• JSON JSONObject和JSONArray共同实现的接口类，JSONUtil.parse方法默认返回此对象（因为不知道是JSON对象还是JSON数组），然后可以根据实际类型判断后转换对象类型。

与FastJSON类似，JSONObject实现了Map接口，JSONArray实现了List接口，这样我们便可以使用熟悉的API来操作JSON。

在JSON中，Hutool封装了getXXX方法，支持大部分内置类型的值获取操作。比如：

```
JSONObject json1 = JSONUtil.createObj();
json1.getStr("key");
json1.getInt("key");
json1.getLong("key");
json1.getDouble("key");
json1.getBigDecimal("key");
```

这些成员方法的加入，可以省掉大量的类型转换代码，大大提高JSON的操作简便性。

JSONObject

介绍

JSONObject代表一个JSON中的键值对象，这个对象以大括号包围，每个键值对使用,隔开，键与值使用:隔开，一个JSONObject类似于这样：

```
{
  "key1":"value1",
  "key2":"value2"
}
```

此处键部分可以省略双引号，值为字符串时不能省略，为数字或波尔值时不加双引号。

使用

创建

```
JSONObject json1 = JSONUtil.createObj();
```

```
json1.put("a", "value1");
json1.put("b", "value2");
json1.put("c", "value3");
```

JSONUtil.createObj()是快捷新建JSONObject的工具方法，同样我们可以直接new：

```
JSONObject json1 = new JSONObject();
```

转换

```
String jsonStr = "{\"b\":\"value2\",\"c\":\"value3\",\"a\":\"value1\"}";
//方法一：使用工具类转换
JSONObject jsonObject = JSONUtil.parseObj(jsonStr);
//方法二：new的方式转换
JSONObject jsonObject2 = new JSONObject(jsonStr);

//JSON对象转字符串
jsonObject.toString();
```

同样，JSONUtil还可以支持以下对象转为JSONObject对象：

- String对象
- Java Bean对象
- Map对象
- XML字符串（使用JSONUtil.parseFromXml方法）
- ResourceBundle(使用JSONUtil.parseFromResourceBundle)

JSONUtil还提供了JSONObject对象转换为其它对象的方法：

- toJsonStr 转换为JSON字符串
- toXmlStr 转换为XML字符串
- toBean 转换为JavaBean

-

JSONArray

介绍

在JSON中，JSONArray代表一个数组，使用中括号包围，每个元素使用逗号隔开。一个JSONArray类似于这样：

```
["value1","value2","value3"]
```

使用

创建

```
//方法1
JSONArray array = JSONUtil.createArray();
//方法2
JSONArray array = new JSONArray();

array.add("value1");
array.add("value2");
array.add("value3");

//转为JSONArray字符串
array.toString();
```

转换

```
String jsonStr = "[\"value1\", \"value2\", \"value3\"]";
JSONArray array = JSONUtil.parseArray(jsonStr);
```

JSONUtil

介绍

JSONUtil是针对JSONObject和JSONArray的静态快捷方法集合，在之前的章节我们已经介绍了一些工具方法，在本章节我们将做一些补充。

使用

parseXXX和toXXX

这两种方法主要是针对JSON和其它对象之间的转换。

readXXX

这类方法主要是从JSON文件中读取JSON对象的快捷方法。包括：

- readJSON
- readJSONObject
- readJSONArray

其它方法

除了上面中常用的一些方法，JSONUtil还提供了一些JSON辅助方法：

• quote 对所有双引号做转义处理（使用双反斜杠做转义）

• wrap 包装对象，可以将普通任意对象转为JSON对象

• formatJsonStr 格式化JSON字符串，此方法并不严格检查JSON的格式正确与否

加密解密(Hutool-crypto)

概述

Hutool-crypto概述

加密分为三种：

1. 对称加密（symmetric），例如：AES、DES等

2. 非对称加密（asymmetric），例如：RSA、DSA等

3. 摘要加密（digest），例如：MD5、SHA-1、SHA-256、HMAC等

hutool-crypto针对这三种加密类型分别封装，并提供常用的大部分加密算法。

对于非对称加密，实现了：

• RSA

• DSA

对于对称加密，实现了：

• AES

• ARCFOUR

• Blowfish

• DES

• DESede

• RC2

• PBEWithMD5AndDES

• PBEWithSHA1AndDESede

• PBEWithSHA1AndRC2_40

对于摘要算法实现了：

• MD2

• MD5

• SHA-1

• SHA-256

• SHA-384

• SHA-512

- HmacMD5
- HmacSHA1
- HmacSHA256
- HmacSHA384
- HmacSHA512

其中，针对常用到的算法，模块还提供SecureUtil工具类用于快速实现加密。

关于各种加密方式的使用，请参阅后续章节。

对称加密-SymmetricCrypto

介绍

对称加密(也叫私钥加密)指加密和解密使用相同密钥的加密算法。有时又叫传统密码算法，就是加密密钥能够从解密密钥中推算出来，同时解密密钥也可以从加密密钥中推算出来。而在大多数的对称算法中，加密密钥和解密密钥是相同的，所以也称这种加密算法为秘密密钥算法或单密钥算法。它要求发送方和接收方在安全通信之前，商定一个密钥。对称算法的安全性依赖于密钥，泄漏密钥就意味着任何人都可以对他们发送或接收的消息解密，所以密钥的保密性对通信的安全性至关重要。

对于对称加密，封装了JDK的，具体介绍见：https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#KeyGenerator(https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#KeyGenerator)：

- AES (默认AES/ECB/PKCS5Padding)
- ARCFOUR
- Blowfish
- DES (默认DES/ECB/PKCS5Padding)
- DESede
- RC2
- PBEWithMD5AndDES
- PBEWithSHA1AndDESede
- PBEWithSHA1AndRC2_40

使用

通用使用

以AES算法为例：

```
String content = "test中文";
```

```
//随机生成密钥
byte[] key =
SecureUtil.generateKey(SymmetricAlgorithm.AES.getValue()).getEncoded();

//构建
SymmetricCrypto aes = new SymmetricCrypto(SymmetricAlgorithm.AES, key);

//加密
byte[] encrypt = aes.encrypt(content);
//解密
byte[] decrypt = aes.decrypt(encrypt);

//加密为16进制表示
String encryptHex = aes.encryptHex(content);
//解密为字符串
String decryptStr = aes.decryptStr(encryptHex, CharsetUtil.CHARSET_UTF_8);
```

DESede实现

```
String content = "test中文";

byte[] key =
SecureUtil.generateKey(SymmetricAlgorithm.DESede.getValue()).getEncoded();

SymmetricCrypto des = new SymmetricCrypto(SymmetricAlgorithm.DESede, key);

//加密
byte[] encrypt = des.encrypt(content);
//解密
byte[] decrypt = des.decrypt(encrypt);

//加密为16进制字符串（Hex表示）
String encryptHex = des.encryptHex(content);
//解密为字符串
String decryptStr = des.decryptStr(encryptHex);
```

AES封装

AES全称高级加密标准（英语：Advanced Encryption Standard，缩写：AES），在密码学中又称Rijndael加密法。

对于Java中AES的默认模式是：AES/ECB/PKCS5Padding，如果使用CryptoJS，请调整为

：padding: CryptoJS.pad.Pkcs7

1. 快速构建

```
String content = "test中文";

// 随机生成密钥
byte[] key =
SecureUtil.generateKey(SymmetricAlgorithm.AES.getValue()).getEncoded();

// 构建
AES aes = SecureUtil.aes(key);

// 加密
byte[] encrypt = aes.encrypt(content);
// 解密
byte[] decrypt = aes.decrypt(encrypt);

// 加密为16进制表示
String encryptHex = aes.encryptHex(content);
// 解密为字符串
String decryptStr = aes.decryptStr(encryptHex, CharsetUtil.CHARSET_UTF_8);
```

1. 自定义模式和偏移

```
AES aes = new AES(Mode.CTS, Padding.PKCS5Padding,
"0CoJUm6Qyw8W8jud".getBytes(), "0102030405060708".getBytes());
```

DES封装

DES全称为Data Encryption Standard，即数据加密标准，是一种使用密钥加密的块算法，Java中默认实现为：DES/CBC/PKCS5Padding

DES使用方法与AES一致，构建方法为：

1. 快速构建

```
byte[] key =
SecureUtil.generateKey(SymmetricAlgorithm.DES.getValue()).getEncoded();
DES des = SecureUtil.des(key);
```

1. 自定义模式和偏移

```
DES des = new DES(Mode.CTS, Padding.PKCS5Padding,
"0CoJUm6Qyw8W8jud".getBytes(), "01020304".getBytes());
```

非对称加密-AsymmetricCrypto

介绍

对于非对称加密，最常用的就是RSA和DSA，在Hutool中使用AsymmetricCrypto对象来负责加密解密。

非对称加密有公钥和私钥两个概念，私钥自己拥有，不能给别人，公钥公开。根据应用的不同，我们可以选择使用不同的密钥加密：

1. 签名：使用私钥加密，公钥解密。用于让所有公钥所有者验证私钥所有者的身份并且用来防止私钥所有者发布的内容被篡改，但是不用来保证内容不被他人获得。

2. 加密：用公钥加密，私钥解密。用于向公钥所有者发布信息,这个信息可能被他人篡改,但是无法被他人获得。

Hutool封装了JDK的，详细见https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#KeyPairGenerator(https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#KeyPairGenerator)：

- RSA
- DSA
- EC

使用

在非对称加密中，我们可以通过AsymmetricCrypto(AsymmetricAlgorithm algorithm)构造方法，通过传入不同的算法枚举，获得其加密解密器。

当然，为了方便，我们针对最常用的RSA和DSA算法构建了单独的对象：RSA和DSA。

基本使用

我们以RSA为例，介绍使用RSA加密和解密 在构建RSA对象时，可以传入公钥或私钥，当使用无参构造方法时，Hutool将自动生成随机的公钥私钥密钥对：

```
RSA rsa = new RSA();

//获得私钥
rsa.getPrivateKey()
rsa.getPrivateKeyBase64()
//获得公钥
rsa.getPublicKey()
rsa.getPublicKeyBase64()

//公钥加密，私钥解密
byte[] encrypt = rsa.encrypt(StrUtil.bytes("我是一段测试aaaa",
CharsetUtil.CHARSET_UTF_8), KeyType.PublicKey);
```

```
byte[] decrypt = rsa.decrypt(encrypt, KeyType.PrivateKey);

//Junit单元测试
//Assert.assertEquals("我是一段测试aaaa", StrUtil.str(decrypt,
CharsetUtil.CHARSET_UTF_8));

//私钥加密，公钥解密
byte[] encrypt2 = rsa.encrypt(StrUtil.bytes("我是一段测试aaaa",
CharsetUtil.CHARSET_UTF_8), KeyType.PrivateKey);
byte[] decrypt2 = rsa.decrypt(encrypt2, KeyType.PublicKey);

//Junit单元测试
//Assert.assertEquals("我是一段测试aaaa", StrUtil.str(decrypt2,
CharsetUtil.CHARSET_UTF_8));
```

> 对于加密和解密可以完全分开，对于RSA对象，如果只使用公钥或私钥，另一个参数可以为null

自助生成密钥对

有时候我们想自助生成密钥对可以：

```
KeyPair pair = SecureUtil.generateKeyPair("RSA");
pair.getPrivate();
pair.getPublic();
```

自助生成的密钥对是byte[]形式，我们可以使用Base64.encode方法转为Base64，便于存储为文本。

当然，如果使用RSA对象，也可以使用encryptStr和decryptStr加密解密为字符串。

案例

案例一：

已知私钥和密文，如何解密密文？

```
String PRIVATE_KEY =
"MIICdQIBADANBgkqhkiG9w0BAQEFAASCAl8wggJbAgEAAoGBAIL7pbQ+5KKGYRhw7j
E31hmA"
+
"f8Q60ybd+xZuRmuO5kOFBRqXGxKTQ9TfQI+aMW+0lw/kibKzaD/EKV91107xE384qOy
6IcuBfaR5lv39OcoqNZ"
+
```

```
"5l+Dah5ABGnVkBP9fKOFhPgghBknTRo0/rZFGI6Q1UHXb+4atP++LNFlDymJcPAgMB
AAECgYBammGb1alndta"
+
"xBmTtLLdveoBmp14p04D8mhkiC33iFKBcLUvvxGg2Vpuc+cbagyu/NZG+R/WDrlgEDU
p6861M5BeFN0L9O4hz"
+
"GAEn8xyTE96f8sh4VlRmBOvVdwZqRO+ilkOM96+KL88A9RKdp8V2tna7TM6oI3LHDyf/
JBoXaQJBAMcVN7fKlYP"
+
"Skzfh/yZzW2fmC0ZNg/qaW8Oa/wfDxlWjgnS0p/EKWZ8BxjR/d199L3i/KMaGdfpaWbY
ZLvYENqUCQQCobjsuCW"
+
"nlZhcWajjzpsSuy8/bICVEpUax1fUZ58Mq69CQXfaZemD9Ar4omzuEAAs2/uee3kt3AvC
Baeq05NyjAkBme8SwB0iK"
+
"kLcaeGuJlq7CQIkjSrobIqUEf+CzVZPe+AorG+isS+Cw2w/2bHu+G0p5xSYvdH59P0+ZT
0N+f9LFAkA6v3Ae56OrI"
+
"wfMhrJksfeKbIaMjNLS9b8JynIaXg9iCiyOHmgkMl5gAbPoH/ULXqSKwzBw5mJ2GW1gB
lyaSfV3AkA/RJC+adIjsRGg"
+
"JOkiRjSmPpGv3FOhl9fsBPjupZBEIuoMWOC8GXK/73DHxwmfNmN7C9+sIi4RBcjEeQ5F
5FHZ";

RSA rsa = new RSA(PRIVATE_KEY, null);

String a =
"2707F9FD4288CEF302C972058712F24A5F3EC62C5A14AD2FC59DAB93503AA0FA171
13A020EE4EA35EB53F"
+
"75F36564BA1DABAA20F3B90FD39315C30E68FE8A1803B36C29029B23EB612C06ACF3
A34BE815074F5EB5AA3A"
+
"C0C8832EC42DA725B4E1C38EF4EA1B85904F8B10B2D62EA782B813229F9090E6F739
4E42E6F44494BB8";

byte[] aByte = HexUtil.decodeHex(a);
byte[] decrypt = rsa.decrypt(aByte, KeyType.PrivateKey);

//Junit单元测试
//Assert.assertEquals("虎头闯杭州,多抬头看天,切勿只管种地", StrUtil.str(decrypt,
CharsetUtil.CHARSET_UTF_8));
```

签名和验证-Sign

介绍

Hutool针对java.security.Signature做了简化包装，包装类为：Sign，用于生成签名和签名验证。

对于签名算法，Hutool封装了JDK的，具体介绍见：https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#Signature(https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#Signature)：

```
// The RSA signature algorithm
NONEwithRSA

// The MD2/MD5 with RSA Encryption signature algorithm
MD2withRSA
MD5withRSA

// The signature algorithm with SHA-* and the RSA
SHA1withRSA
SHA256withRSA
SHA384withRSA
SHA512withRSA

// The Digital Signature Algorithm
NONEwithDSA

// The DSA with SHA-1 signature algorithm
SHA1withDSA

// The ECDSA signature algorithms
NONEwithECDSA
SHA1withECDSA
SHA256withECDSA
SHA384withECDSA
SHA512withECDSA
```

使用

```
byte[] data = "我是一段测试字符串".getBytes();
Sign sign = SecureUtil.sign(SignAlgorithm.MD5withRSA);
//签名
byte[] signed = sign.sign(data);
```

```
//验证签名
boolean verify = sign.verify(data, signed);
```

摘要加密-Digester和HMac

介绍

摘要算法介绍

摘要算法是一种能产生特殊输出格式的算法，这种算法的特点是：无论用户输入什么长度的原始数据，经过计算后输出的密文都是固定长度的，这种算法的原理是根据一定的运算规则对原数据进行某种形式的提取，这种提取就是摘要，被摘要的数据内容与原数据有密切联系，只要原数据稍有改变，输出的“摘要”便完全不同，因此，基于这种原理的算法便能对数据完整性提供较为健全的保障。

但是，由于输出的密文是提取原数据经过处理的定长值，所以它已经不能还原为原数据，即消息摘要算法是不可逆的，理论上无法通过反向运算取得原数据内容，因此它通常只能被用来做数据完整性验证。

HMAC介绍

HMAC，全称为“Hash Message Authentication Code”，中文名“散列消息鉴别码”，主要是利用哈希算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出。一般的，消息鉴别码用于验证传输于两个共 同享有一个密钥的单位之间的消息。HMAC 可以与任何迭代散列函数捆绑使用。MD5 和 SHA-1 就是这种散列函数。HMAC 还可以使用一个用于计算和确认消息鉴别值的密钥。

Hutool支持的摘要算法类型

详细见
：https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#MessageDigest(https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#MessageDigest)

摘要算法

- MD2
- MD5
- SHA-1
- SHA-256
- SHA-384
- SHA-512

Hmac算法

- HmacMD5
- HmacSHA1
- HmacSHA256
- HmacSHA384
- HmacSHA512

摘要算法抽象

摘要对象被抽象为两个对象：

- Digester
- HMac

使用

Digester

以MD5为例：

```
Digester md5 = new Digester(DigestAlgorithm.MD5);
String digestHex = md5.digestHex(testStr);
//Junit单元测试
//Assert.assertEquals("5393554e94bf0eb6436f240a4fd71282", digestHex);
```

当然，做为最为常用的方法，MD5等方法被封装为工具方法在DigestUtil中，以上代码可以进一步简化为：

```
String md5Hex1 = DigestUtil.md5Hex(testStr);
//Junit单元测试
//Assert.assertEquals("5393554e94bf0eb6436f240a4fd71282", md5Hex1);
```

HMac

以HmacMD5为例：

```
String testStr = "test中文";

byte[] key = "password".getBytes();
HMac mac = new HMac(HmacAlgorithm.HmacMD5, key);

String macHex1 = mac.digestHex(testStr);
//Junit单元测试
```

```
//Assert.assertEquals("b977f4b13f93f549e06140971bded384", macHex1);
```

加密解密工具-SecureUtil

介绍

SecureUtil主要针对常用加密算法构建快捷方式，还有提供一些密钥生成的快捷工具方法。

方法介绍

对称加密

- SecureUtil.aes
- SecureUtil.des

摘要算法

- SecureUtil.md5
- SecureUtil.sha1
- SecureUtil.hmac
- SecureUtil.hmacMd5
- SecureUtil.hmacSha1

非对称加密

- SecureUtil.rsa
- SecureUtil.dsa

UUID

- SecureUtil.simpleUUID 方法提供无“-”的UUID

密钥生成

- SecureUtil.generateKey 针对对称加密生成密钥
- SecureUtil.generateKeyPair 生成密钥对（用于非对称加密）
- SecureUtil.generateSignature 生成签名（用于非对称加密）

其它方法为针对特定加密方法的一些密钥生成和签名相关方法，详细请参阅API文档。

DFA查找(Hutool-dfa)

概述

由来

在我最早入职的一家公司，主要负责内容方面的业务，对我来说大部分的工作是对内容的清洗和规整。当然，清洗过程免不了的就是按照关键词过滤，你懂的。需求如下：

后台人员添加N个关键字，然后对主站所有的内容进行清洗，含有这些关键字的所有内容都置为无效。

思路

拿到此需求，我最早的方案比较粗暴：针对关键字建立一个HashSet，然后遍历整个数据库，针对每篇文章遍历这个Set，查找是否contains关键字……好吧我承认这不是一个好方法，随着关键字的增多和数据的增多，这个过程消耗的时间成指数型增长！

于是我找到度娘，发现一个算法：DFA。

DFA介绍

DFA全称为：Deterministic Finite Automaton,即确定有穷自动机。因为本人算法学的不好，有兴趣的可以看这篇博客: 基于DFA敏感词查询的算法简析(http://www.cnblogs.com/naaoveGIS/archive/2016/10/14/5960352.html)

解释起来原理其实也不难，就是用所有关键字构造一棵树，然后用正文遍历这棵树，遍历到叶子节点即表示文章中存在这个关键字。

我们暂且忽略构建关键词树的时间，每次查找正文只需要O(n)复杂度就可以搞定。

针对DFA算法以及网上的一些实现，Hutool做了整理和改进，最终形成现在的Hutool-dfa模块。

DFA查找

使用

1. 构建关键词树

```
WordTree tree = new WordTree();
tree.addWord("大");
tree.addWord("大土豆");
tree.addWord("土豆");
tree.addWord("刚出锅");
tree.addWord("出锅");
```

2. 查找关键词

```
//正文
String text = "我有一颗大土豆，刚出锅的";
```

1. 情况一：标准匹配，匹配到最短关键词，并跳过已经匹配的关键词

```
// 匹配到【大】，就不再继续匹配了，因此【大土豆】不匹配
// 匹配到【刚出锅】，就跳过这三个字了，因此【出锅】不匹配（由于刚首先被匹配，因此长的被匹配，最短匹配只针对第一个字相同选最短）
List<String> matchAll = tree.matchAll(text, -1, false, false);
Assert.assertEquals(matchAll.toString(), "[大, 土豆, 刚出锅]");
```

1. 情况二：匹配到最短关键词，不跳过已经匹配的关键词

```
// 【大】被匹配，最短匹配原则【大土豆】被跳过，【土豆继续被匹配】
// 【刚出锅】被匹配，由于不跳过已经匹配的词，【出锅】被匹配
matchAll = tree.matchAll(text, -1, true, false);
Assert.assertEquals(matchAll.toString(), "[大, 土豆, 刚出锅, 出锅]");
```

1. 情况三：匹配到最长关键词，跳过已经匹配的关键词

```
// 匹配到【大】，由于到最长匹配，因此【大土豆】接着被匹配
// 由于【大土豆】被匹配，【土豆】被跳过，由于【刚出锅】被匹配，【出锅】被跳过
matchAll = tree.matchAll(text, -1, false, true);
Assert.assertEquals(matchAll.toString(), "[大, 大土豆, 刚出锅]");
```

1. 情况四：匹配到最长关键词，不跳过已经匹配的关键词（最全关键词）

```
// 匹配到【大】，由于到最长匹配，因此【大土豆】接着被匹配，由于不跳过已经匹配的关键词，土豆继续被匹配
// 【刚出锅】被匹配，由于不跳过已经匹配的词，【出锅】被匹配
matchAll = tree.matchAll(text, -1, true, true);
Assert.assertEquals(matchAll.toString(), "[大, 大土豆, 土豆, 刚出锅, 出锅]");
```

> 除了matchAll方法，WordTree还提供了match和isMatch两个方法，这两个方法只会查找第一个匹配的结果，这样一旦找到第一个关键字，就会停止继续匹配，大大提高了匹配效率。

针对特殊字符

有时候，正文中的关键字常常包含特殊字符，比如:"〓关键☆字"，针对这种情况，Hutool提供了StopChar类，专门针对特殊字符做跳过处理，这个过程是在match方法或matchAll方法执行的时候自动去掉特殊字符。

数据库(Hutool-db)

概述

由来

Hutool-db是一个在JDBC基础上封装的数据库操作工具类，通过包装，使用ActiveRecord思想操作数据库。在Hutool-db中，使用Entity（本质上是个Map）代替Bean来使数据库操作更加灵活，同时提供Bean和Entity的转换提供传统ORM的兼容支持。

整体的架构

整体分为几部分:

图片地址：https://static.oschina.net/uploads/img/201712/28150856_Yu1z.png

1. 数据源 DataSource
2. SQL执行器 SqlExecutor
3. CRUD的封装 Db、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：

CRUD的封装 Db 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。

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. EntityListHandler 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. EntityHandler 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。

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，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：

```
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);
}
```

数据库简单操作-Db

由来

数据库操作不外乎四门功课：增删改查，在Java的世界中，由于JDBC的存在，这项工作变得简单易用，但是也并没有做到使用上的简化。于是出现了JPA（Hibernate）、MyBatis、Jfinal等解决框架，或解决多数据库差异问题，或解决SQL维护问题。而Hutool对JDBC的封装，多数为在小型项目中对数据处理的简化，尤其只涉及单表操作时。OK，废话不多，来个Demo感受下。

使用

我们以MySQL为例

1、添加配置文件

Maven项目中在src/main/resources目录下添加db.setting文件（非Maven项目添加到ClassPath中即可）：

```
## db.setting文件

url = jdbc:mysql://localhost:3306/test
user = root
pass = 123456

## 可选配置
# 是否在日志中显示执行的SQL
showSql = true
# 是否格式化显示的SQL
formatSql = false
# 是否显示SQL参数
showParams = true
```

2、引入MySQL JDBC驱动jar

```
<!--mysql数据库驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${mysql.version}</version>
</dependency>
```

> 注意
>
> 此处不定义MySQL版本，请参考官方文档使用匹配的驱动包版本。

3、增删改查

增

```
Db.use().insert(
    Entity.create("user")
    .set("name", "unitTestUser")
    .set("age", 66)
```

```
);
```

插入数据并返回自增主键：

```
Db.use().insertForGeneratedKey(
    Entity.create("user")
    .set("name", "unitTestUser")
    .set("age", 66)
);
```

删

```
Db.use().del(
    Entity.create("user").set("name", "unitTestUser")//where条件
);
```

改

```
Db.use().update(
    Entity.create().set("age", 88), //修改的数据
    Entity.create("user").set("name", "unitTestUser") //where条件
);
```

> 注意
>
> 条件语句除了可以用=精确匹配外，也可以范围条件匹配，例如表示 age < 12 可以这样构造Entity：Entity.create("user").set("age", "< 12")，但是通过Entity方式传入条件暂时不支持同字段多条件的情况。

查

1. 查询全部字段

```
//user为表名
Db.use().findAll("user");
```

1. 条件查询

```
Db.use().findAll(Entity.create("user").set("name", "unitTestUser"));
```

1. 模糊查询

```
Db.use().findLike("user", "name", "Test", LikeType.Contains);
```

1. 分页查询

```
//Page对象通过传入页码和每页条目数达到分页目的
PageResult<Entity> result = Db.use().page(Entity.create("user").set("age", "> 30"), new Page(10, 20));
```

1. 执行SQL语句

```
//查询
List<Entity> = Db.use().query("select * from user where age < ?", 3);
```

```
//更新
Db.use().query("update user set age = ? where name = ?", 3, "张三");
```

1. 事务

```
Db.use().tx(new TxFunc() {
    @Override
    public void call(Db db) throws SQLException {
        db.insert(Entity.create("user").set("name", "unitTestUser"));
        db.update(Entity.create().set("age", 79), Entity.create("user").set("name",
"unitTestUser"));
    }
});
```

数据源工厂-DsFactory

释义

数据源（DataSource）的概念来自于JDBC规范中，一个数据源表示针对一个数据库（或者集群）的描述，从数据源中我们可以获得N个数据库连接，从而对数据库进行操作。

每一个开源JDBC连接池都有对DataSource的实现，比如Druid为DruidDataSource，Hikari为HikariDataSource。但是各大连接池配置各不相同，配置文件也不一样，Hutool的针对常用的连接池做了封装，最大限度简化和提供一致性配置。

Hutool的解决方案是：在ClassPath中使用config/db.setting一个配置文件，配置所有种类连接池的数据源，然后使用DsFactory.get()方法自动识别数据源以及自动注入配置文件中的连接池配置（包括数据库连接配置）。DsFactory通过try的方式按照顺序检测项目中引入的jar包来甄别用户使用的是哪种连接池，从而自动构建相应的数据源。

Hutool支持以下连接池，并按照其顺序检测存在与否：

1. HikariCP
2. Druid
3. Tomcat
4. Dbcp
5. C3p0

在没有引入任何连接池的情况下，Hutool会使用其内置的连接池：Hutool Pooled（简易连接池，不推荐在线上环境使用）。

基本使用

1. 引入连接池的jar

Hutool不会强依赖于任何第三方库，在Hutool支持的连接池范围内，用户需自行选择自己喜欢的连接池并引入。

2. 编写配置文件

Maven项目中，在src/main/resources/config下创建文件db.setting，编写配置文件即可。这个配置文件位置就是Hutool与用户间的一个约定（符合约定大于配置的原则）：

配置文件分为两部分

1. 基本连接信息

```
## 基本配置信息
# JDBC URL，根据不同的数据库，使用相应的JDBC连接字符串
url = jdbc:mysql://<host>:<port>/<database_name>
# 用户名，此处也可以使用 user 代替
username = 用户名
# 密码，此处也可以使用 pass 代替
password = 密码
# JDBC驱动名，可选（Hutool会自动识别）
driver = com.mysql.jdbc.Driver
```

> **小提示**
>
> 其中driver是可选的，Hutool会根据url自动加载相应的Driver类。基本连接信息是所有连接池通用的，原则上，只有基本信息就可以成功连接并操作数据库。

2. 连接池特有配置信息

针对不同的连接池，除了基本信息外的配置都各不相同，Hutool针对不同的连接池封装了其配置项，可以在项目的src/test/resources/example中看到针对不同连接池的配置文件样例。

我们以HikariCP为例：

```
# 自动提交
autoCommit = true
# 等待连接池分配连接的最大时长（毫秒），超过这个时长还没可用的连接则发生
SQLException， 缺省:30秒
```

```
connectionTimeout = 30000
# 一个连接idle状态的最大时长（毫秒），超时则被释放（retired），缺省:10分钟
idleTimeout = 600000
# 一个连接的生命时长（毫秒），超时而且没被使用则被释放（retired），缺省:30分钟，建议设置比数据库超时时长少30秒，参考MySQL wait_timeout参数（show variables like '%timeout%';）
maxLifetime = 1800000
# 获取连接前的测试SQL
connectionTestQuery = SELECT 1
# 最小闲置连接数
minimumIdle = 10
# 连接池中允许的最大连接数。缺省值：10；推荐的公式：((core_count * 2) + effective_spindle_count)
maximumPoolSize = 10
# 连接只读数据库时配置为true， 保证安全
readOnly = false
```

3. 获取数据源

```
//获取默认数据源
DataSource ds = DSFactory.get()
```

是滴，就是这么简单，一个简单的方法，可以识别数据源并读取默认路径(config/db.setting)下信息从而获取数据源。

4. 直接创建数据源

当然你依旧可以按照连接池本身的方式获取数据源对象。我们以Druid为例：

```
//具体的配置参数请参阅Druid官方文档
DruidDataSource ds2 = new DruidDataSource();
ds2.setUrl("jdbc:mysql://localhost:3306/dbName");
ds2.setUsername("root");
ds2.setPassword("123456");
```

5. 创建简单数据源

有时候我们的操作非常简单，亦或者只是测试下远程数据库是否畅通，我们可以使用Hutool提供的SimpleDataSource:

```
DataSource ds = new SimpleDataSource("jdbc:mysql://localhost:3306/dbName", "root", "123456");
```

SimpleDataSource只是DriverManager.getConnection的简单包装，本身并不支持池化功能，此类特别适合少量数据库连接的操作。

同样的，SimpleDataSource也支持默认配置文件：

```
DataSource ds = new SimpleDataSource();
```

高级实用

1. 自定义连接池

有时候当项目引入多种数据源时，我们希望自定义需要的连接池，此时可以：

```
//自定义连接池实现为Tomcat-pool
DSFactory.setCurrentDSFactory(new TomcatDSFactory());
DataSource ds = DSFactory.get();
```

需要注意的是，DSFactory.setCurrentDSFactory是一个全局方法，必须在所有获取数据源的时机之前调用，调用一次即可（例如项目启动）。

2. 自定义配置文件

有时候由于项目规划的问题，我们希望自定义数据库配置Setting的位置，甚至是动态加载Setting对象，此时我们可以使用以下方法从其它的Setting对象中获取数据库连接信息：

```
//自定义数据库Setting，更多实用请参阅Hutool-Setting章节
Setting setting = new Setting("otherPath/other.setting");
//获取指定配置，第二个参数为分组，用于多数据源，无分组情况下传null
DataSource ds = DSFactory.get(setting, null);
```

3. 多数据源

有的时候我们需要操作不同的数据库，也有可能我们需要针对线上、开发和测试分别操作其数据库，无论哪种情况，Hutool都针对多数据源做了很棒的支持。

多数据源有两种方式可以实现：

1. 多个配置文件分别获得数据源

就是按照自定义配置文件的方式读取多个配置文件即可。

2. 在同一配置文件中使用分组隔离不同的数据源配置：

```
[group_db1]
url = jdbc:mysql://<host>:<port>/<database_name>
```

```
username = 用户名
password = 密码

[group_db2]
url = jdbc:mysql://<host2>:<port>/<database_name>
username = 用户名
password = 密码
```

我们按照上面的方式编写db.setting文件，然后：

```
DataSource ds1 = DSFactory.get("group_db1");
DataSource ds2 = DSFactory.get("group_db2");
```

这样我们就可以在一个配置文件中实现多数据源的配置。

结语

Hutool通过多种方式获取DataSource对象，获取后除了可以在Hutool自身应用外，还可以将此对象传入不同的框架以实现无缝结合。

Hutool对数据源的封装很好的诠释了以下几个原则：

1. 自动识别优于用户定义
2. 便捷性与灵活性并存
3. 适配与兼容

简单CRUD-SqlRunner

由来

在最基本的数据库编码中，最常用的要数CRUD（既增查改删，也叫增删改查）。在Hutoo-db模块中，把这些逻辑全部封装于SqlRunner中。

使用

1. 创建SqlRunner

在上一章节中，我们详细的介绍了如何获取DataSource对象，而SqlRunner的构造就是依赖于DataSource:

```
DataSource ds = DSFactory.get();
SqlRunner runner = SqlRunner.create(ds);
```

非常简单的两句就可以构造SqlRunner对象，当然，Hutool还针对默认配置文件，对上面的代码进一步简化：

```
SqlRunner runner = SqlRunner.create();
```

这就是约定大于配置的优势：代码可以做到及其精简。

2. 对数据库操作

在SqlRunner中，提供了丰富的方法来操作数据库的增删改查。

需要说明的是，Hutool-db吸取了Jfinal的ActiveRecord思想，并从Python语言的Django框架中学习部分思想，因此SqlRunner中操作数据库传入的数据和条件都为Entity对象——一个继承于Map的动态K-V对象。这个对象中key为字段名，value为字段值。通过这种方式，我们读取和插入的数据可以不考虑类型（Hutool会自动识别并处理相应类型）。

具体操作如下：

1. 插入数据

```
//runner既SqlRunner对象
int count = runner.insert(Entity.create("user")
    .set("name", "unitTestUser")
    .set("age", 66));
```

以上我们构建了一个Entity对象，对应数据库中的user表，set入各字段值。

2. 插入数据并返回主键

很多时候我们的表主键为自增主键，每次插入希望返回主键，我们可以：

```
Long id = runner.insertForGeneratedKey(Entity.create("user")
    .set("name", "张三")
    .set("age", 66));
```

3. 批量插入

```
User user1 = new User();
user1.setName("张三");
user1.setAge(12);
user1.setBirthday("19900112");
user1.setGender(true);

User user2 = new User();
user2.setName("李四");
user2.setAge(12);
user2.setBirthday("19890512");
```

```
user2.setGender(false);

Entity data1 = Entity.parse(user1);
Entity data2 = Entity.parse(user2);

int[] result = runner.insert(CollectionUtil.newArrayList(data1, data2));
```

需要注意的是，批量插入每一条数据结构必须一致。批量插入数据时会获取第一条数据的字段结构，之后的数据会按照这个格式插入。也就是说假如第一条数据只有2个字段，后边数据多于这两个字段的部分将被抛弃。

4. 修改数据

```
int update = runner.update(
    Entity.create().set("age", 88),
    Entity.create("user").set("name", "unitTestUser")
);
```

修改数据我们构建了两个Entity对象，第一个表示要修改的数据项和数据值，第二个表示where条件（要修改哪些数据）。表名可以在两个Entity对象中的任意一个中设置。最终生成的SQL：

```
UPDATE user SET age = ? WHERE name = ?
```

值通过PreparedStatemnt方式注入（最大限度的避免SQL注入）

> **小提示**
>
> 条件语句除了可以用=精确匹配外，也可以范围条件匹配，例如表示 age < 12 可以这样构造Entity：Entity.create("user").set("age", "< 12")，但是通过Entity方式传入条件暂时不支持同字段多条件的情况。

5. 删除数据

```
int delCount = runner.del(Entity.create("user").set("name", "unitTestUser"));
```

同样的，删除的条件也支持范围条件匹配。最终生成的SQL：

```
DELETE FROM user WHERE name = ?
```

> **小提示**
>
> 为了安全，Hutool并不支持删除操作传入空条件（既生成SQL后无WHERE条件），这样可以一定程度上避免传入null导致全表清空的事故。

6. 数据查询

数据查询主要针对条件查询、分页查询等功能的封装，Hutool查询支持以下几种数据库方言：

- MySQL
- Sqllite
- Oracle
- PostgreSQL

当然其它数据库例如SQLServer、H2、Derby等数据库也同样支持，只是并未实现其特有方言，针对标准的CRUD，则会按照ANSI标准生成SQL语句。

(1). 条件查询

find方法可以按照指定条件查询数据库，方法定义如下：

```
T find(Collection<String> fields, Entity where, RsHandler<T> rsh)
```

第一个参数表示返回的字段列表，第二个参数为条件，第三个参数为结果集处理类（JDBC返回ResultSet，此对象定义了ResultSet转换为其它对象的逻辑）

字段列表可以传入null表示查询所有字段（相当于select * ），条件为Entity，与其它方法的条件方法一致。

结果集Hutool提供了几种默认的结果集：

- EntityHandler Entity对象处理器，只处理第一条数据。既当只需要一条数据时，可以使用此处理器，返回一个Entity对象。
- EntityListHandler 返回Entity列表，用于返回多条数据
- EntitySetHandler 返回Entity列表，用于返回多条不重复数据
- NumberHandler 处理为数字结果，当查询结果为单个数字时使用此处理器（例如select count(1)）
- PageResultHandler 分页结果集处理类 ，处理出的结果为PageResult（PageResult中有：页码、每页结果数、总页数、总数等信息）

栗子：

```
runner.find(
    CollectionUtil.newArrayList("name", "age"),
    Entity.create("user").set("name", "unitTestUser"),
    new EntityListHandler()
);
```

(2). 查询全部字段

```
runner.find(
    Entity.create("user").set("name", "unitTestUser"),
    new EntityListHandler()
);
```

对于EntityListHandler处理方式还有快捷方法：

```
runner.findAll(Entity.create("user").set("name", "unitTestUser"));
```

查询全部数据(数据量大慎用)：

```
runner.findAll("user");
```

(3). 单条件查询

```
runner.findBy("user", "name", "unitTestUser");
```

(4). 模糊查询

```
runner.findLike("user", "name", "Test", LikeType.Contains);
```

这句生成的SQL语句为：

```
SELECT * FROM user WHERE name like '%Test%'
```

(5). 分页

分页的使用方法类似于find方法，只是多了页码(page)和每页条目数(numPerPage)两个参数用于控制分页，Hutool的分页支持MySQL、Sqllite、Oracle等主流数据库。

栗子：

```
//Page对象通过传入页码和每页条目数达到分页目的
PageResult<Entity> result = runner.page(Entity.create("user").set("age", "> 30"), new Page(10, 20));
```

其中PageResult继承自ArrayList，本身是个List可以按照列表操作，同时提供方法获取总页数、总数等信息方便前端生成分页列表。

(6). SQL执行

由于Hutool的封装操作功能有限，并不能满足复杂的SQL执行操作（例如多表联查、同字段多条件等），因此Hutool提供了一系列方法用于执行SQL语句：

- query 通过传入SQL、结果集处理器和SQL参数便捷的执行SQL查询
- execute 执行非查询SQL语句
- executeForGeneratedKey 执行非查询语句并返回主键
- executeBatch 执行批量插入，JDBC批量查询的浅封装，数据列表中的字段值列表格式必须一致

支持事务的CRUD-Session

介绍

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：

Session创建

与SqlRunner类似，Session也可以通过调用create

```
//默认数据源
Session session = Session.create();

//自定义数据源（此处取test分组的数据源）
Session session = Session.create(DSFactory.get("test"));
```

事务CRUD

session.beginTransaction()表示事务开始，调用后每次执行语句将不被提交，只有调用commit方法后才会合并提交，提交或者回滚后会恢复默认的自动提交模式。

1. 新增

```
Entity entity = Entity.create(TABLE_NAME).set("字段1", "值").set("字段2", 2);
try {
session.beginTransaction();
// 增，生成SQL为 INSERT INTO `table_name` SET(`字段1`, `字段2`) VALUES(?,?)
session.insert(entity);
session.commit();
} catch (SQLException e) {
session.quietRollback();
}
```

1. 更新

```
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 {
session.beginTransaction();
// 改，生成SQL为 UPDATE `table_name` SET `字段1` = ?, `字段2` = ? WHERE `条件1` = ?
session.update(entity, where);
session.commit();
} catch (SQLException e) {
session.quietRollback();
}
```

1. 删除

```
Entity where = Entity.create(TABLE_NAME).set("条件1", "条件值");
try {
session.beginTransaction();
// 删，生成SQL为 DELETE FROM `table_name` WHERE `条件1` = ?
session.del(where);
session.commit();
} catch (SQLException e) {
session.quietRollback();
}
```

SQL执行器-SqlExecutor

介绍

这是一个静态类，对JDBC的薄封装，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

使用

```
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 EntityListHandler(), "值");
log.info("{}", entityList);
} catch (SQLException e) {
Log.error(log, e, "SQL error!");
} finally {
DbUtil.close(conn);
```

```
}
```

数据源配置db.setting样例

HikariCP

```
#==========================================================
============
# 数据库配置文件样例
# DsFactory默认读取的配置文件是config/db.setting或db.setting
# db.setting的配置包括两部分：基本连接信息和连接池配置信息。
# 基本连接信息所有连接池都支持，连接池配置信息根据不同的连接池，连接池配置是根据连
接池相应的配置项移植而来
#==========================================================
============

#-------------------------------------------------------------------------------------
## 基本配置信息
# JDBC URL，根据不同的数据库，使用相应的JDBC连接字符串
url = jdbc:mysql://<host>:<port>/<database_name>
# 用户名，此处也可以使用 user 代替
username = 用户名
# 密码，此处也可以使用 pass 代替
password = 密码
# JDBC驱动名，可选（Hutool会自动识别）
driver = com.mysql.jdbc.Driver

## 可选配置
# 是否在日志中显示执行的SQL
showSql = true
# 是否格式化显示的SQL
formatSql = false
# 是否显示SQL参数
showParams = true

#-------------------------------------------------------------------------------------
## 连接池配置项

## ------------------------------------------------- HikariCP
# 自动提交
```

```
autoCommit = true
# 等待连接池分配连接的最大时长（毫秒），超过这个时长还没可用的连接则发生SQLException， 缺省:30秒
connectionTimeout = 30000
# 一个连接idle状态的最大时长（毫秒），超时则被释放（retired），缺省:10分钟
idleTimeout = 600000
# 一个连接的生命时长（毫秒），超时而且没被使用则被释放（retired），缺省:30分钟，建议设置比数据库超时时长少30秒，参考MySQL wait_timeout参数（show variables like '%timeout%';）
maxLifetime = 1800000
# 获取连接前的测试SQL
connectionTestQuery = SELECT 1
# 最小闲置连接数
minimumIdle = 10
# 连接池中允许的最大连接数。缺省值：10；推荐的公式：((core_count * 2) + effective_spindle_count)
maximumPoolSize = 10
# 连接只读数据库时配置为true， 保证安全
readOnly = false
```

Druid

```
#================================================================
# 数据库配置文件样例
# DsFactory默认读取的配置文件是config/db.setting
# db.setting的配置包括两部分：基本连接信息和连接池配置信息。
# 基本连接信息所有连接池都支持，连接池配置信息根据不同的连接池，连接池配置是根据连接池相应的配置项移植而来
#================================================================

#--------------------------------------------------------------------------------
## 基本配置信息
# JDBC URL，根据不同的数据库，使用相应的JDBC连接字符串
url = jdbc:mysql://<host>:<port>/<database_name>
# 用户名，此处也可以使用 user 代替
username = 用户名
# 密码，此处也可以使用 pass 代替
password = 密码
# JDBC驱动名，可选（Hutool会自动识别）
```

```
driver = com.mysql.jdbc.Driver

## 可选配置
# 是否在日志中显示执行的SQL
showSql = true
# 是否格式化显示的SQL
formatSql = false
# 是否显示SQL参数
showParams = true

#-------------------------------------------------------------------------------------
## 连接池配置项

## ------------------------------------------------- Druid
# 初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次
getConnection时
initialSize = 0
# 最大连接池数量
maxActive = 8
# 最小连接池数量
minIdle = 0
# 获取连接时最大等待时间，单位毫秒。配置了maxWait之后， 缺省启用公平锁，并发效率会
有所下降， 如果需要可以通过配置useUnfairLock属性为true使用非公平锁。
maxWait = 0
# 是否缓存preparedStatement，也就是PSCache。 PSCache对支持游标的数据库性能提升
巨大，比如说oracle。 在mysql5.5以下的版本中没有PSCache功能，建议关闭掉。作者在
5.5版本中使用PSCache，通过监控界面发现PSCache有缓存命中率记录， 该应该是支持
PSCache。
poolPreparedStatements = false
# 要启用PSCache，必须配置大于0，当大于0时， poolPreparedStatements自动触发修改为
true。 在Druid中，不会存在Oracle下PSCache占用内存过多的问题， 可以把这个数值配置大
一些，比如说100
maxOpenPreparedStatements = -1
# 用来检测连接是否有效的sql，要求是一个查询语句。 如果validationQuery为
null，testOnBorrow、testOnReturn、 testWhileIdle都不会其作用。
validationQuery = SELECT 1
# 申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。
testOnBorrow = true
# 归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能
testOnReturn = false
# 建议配置为true，不影响性能，并且保证安全性。 申请连接的时候检测，如果空闲时间大于
timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效。
```

```
testWhileIdle = false
# 有两个含义： 1) Destroy线程会检测连接的间隔时间 2) testWhileIdle的判断依据，详细看
testWhileIdle属性的说明
timeBetweenEvictionRunsMillis = 60000
# 物理连接初始化的时候执行的sql
connectionInitSqls = SELECT 1
# 属性类型是字符串，通过别名的方式配置扩展插件， 常用的插件有： 监控统计用的
filter:stat  日志用的filter:log4j 防御sql注入的filter:wall
filters = stat
# 类型是List<com.alibaba.druid.filter.Filter>， 如果同时配置了filters和proxyFilters， 是组
合关系，并非替换关系
proxyFilters =
```

Tomcat JDBC Pool

```
#==================================================================
============
# 数据库配置文件样例
# DsFactory默认读取的配置文件是config/db.setting
# db.setting的配置包括两部分：基本连接信息和连接池配置信息。
# 基本连接信息所有连接池都支持，连接池配置信息根据不同的连接池，连接池配置是根据连
接池相应的配置项移植而来
#==================================================================
============

#-------------------------------------------------------------------------------------
## 基本配置信息
# JDBC URL，根据不同的数据库，使用相应的JDBC连接字符串
url = jdbc:mysql://<host>:<port>/<database_name>
# 用户名，此处也可以使用 user 代替
username = 用户名
# 密码，此处也可以使用 pass 代替
password = 密码
# JDBC驱动名，可选（Hutool会自动识别）
driver = com.mysql.jdbc.Driver

## 可选配置
# 是否在日志中显示执行的SQL
showSql = true
# 是否格式化显示的SQL
formatSql = false
```

```
# 是否显示SQL参数
showParams = true

#-------------------------------------------------------------------------------
## 连接池配置项

## ---------------------------------------------------- Tomcat-Jdbc-Pool
# (boolean) 连接池创建的连接的默认的auto-commit 状态
defaultAutoCommit = true
# (boolean) 连接池创建的连接的默认的read-only 状态。 如果没有设置则setReadOnly 方法将不会被调用。（某些驱动不支持只读模式， 比如：Informix)
defaultReadOnly = false
# (String) 连接池创建的连接的默认的TransactionIsolation 状态。 下面列表当中的某一个：( 参考javadoc) NONE READ_COMMITTED EAD_UNCOMMITTED REPEATABLE_READ SERIALIZABLE
defaultTransactionIsolation = NONE
# (int) 初始化连接： 连接池启动时创建的初始化连接数量，1。2 版本后支持
initialSize = 10
# (int) 最大活动连接： 连接池在同一时间能够分配的最大活动连接的数量， 如果设置为非正数则表示不限制
maxActive = 100
# (int) 最大空闲连接： 连接池中容许保持空闲状态的最大连接数量， 超过的空闲连接将被释放， 如果设置为负数表示不限制 如果启用，将定期检查限制连接，如果空闲时间超过 minEvictableIdleTimeMillis 则释放连接 （ 参考testWhileIdle ）
maxIdle = 8
# (int) 最小空闲连接： 连接池中容许保持空闲状态的最小连接数量， 低于这个数量将创建新的连接， 如果设置为0 则不创建 如果连接验证失败将缩小这个值（ 参考testWhileIdle ）
minIdle = 0
# (int) 最大等待时间： 当没有可用连接时， 连接池等待连接被归还的最大时间( 以毫秒计数 )， 超过时间则抛出异常， 如果设置为-1 表示无限等待
maxWait = 30000
# (String) SQL 查询， 用来验证从连接池取出的连接， 在将连接返回给调用者之前。 如果指定， 则查询必须是一个SQL SELECT 并且必须返回至少一行记录 查询不必返回记录，但这样将不能抛出SQL异常
validationQuery = SELECT 1
# (boolean) 指明是否在从池中取出连接前进行检验， 如果检验失败， 则从池中去除连接并尝试取出另一个。注意： 设置为true 后如果要生效，validationQuery 参数必须设置为非空字符串 参考validationInterval以获得更有效的验证
testOnBorrow = false
# (boolean) 指明是否在归还到池中前进行检验 注意： 设置为true 后如果要生效 ，validationQuery 参数必须设置为非空字符串
testOnReturn = false
```

```
# (boolean) 指明连接是否被空闲连接回收器( 如果有) 进行检验。 如果检测失败， 则连接将被从池中去除。注意： 设置为true 后如果要生效，validationQuery 参数必须设置为非空字符串
testWhileIdle = false
```

C3P0（不推荐）

```
#==============================================================
============
# 数据库配置文件样例
# DsFactory默认读取的配置文件是config/db.setting
# db.setting的配置包括两部分：基本连接信息和连接池配置信息。
# 基本连接信息所有连接池都支持，连接池配置信息根据不同的连接池，连接池配置是根据连接池相应的配置项移植而来
#==============================================================
============

#-------------------------------------------------------------------------------------------
## 基本配置信息
# JDBC URL，根据不同的数据库，使用相应的JDBC连接字符串
url = jdbc:mysql://<host>:<port>/<database_name>
# 用户名，此处也可以使用 user 代替
username = 用户名
# 密码，此处也可以使用 pass 代替
password = 密码
# JDBC驱动名，可选（Hutool会自动识别）
driver = com.mysql.jdbc.Driver

## 可选配置
# 是否在日志中显示执行的SQL
showSql = true
# 是否格式化显示的SQL
formatSql = false
# 是否显示SQL参数
showParams = true

#-------------------------------------------------------------------------------------------
## 连接池配置项

## ------------------------------------------------- C3P0
# 连接池中保留的最大连接数。默认值: 15
```

```
maxPoolSize = 15
# 连接池中保留的最小连接数，默认为：3
minPoolSize = 3
# 初始化连接池中的连接数，取值应在minPoolSize与maxPoolSize之间，默认为3
initialPoolSize = 3
# 最大空闲时间，60秒内未使用则连接被丢弃。若为0则永不丢弃。默认值: 0
maxIdleTime = 0
# 当连接池连接耗尽时，客户端调用getConnection()后等待获取新连接的时间，超时后将抛出SQLException，如设为0则无限期等待。单位毫秒。默认: 0
checkoutTimeout = 0
# 当连接池中的连接耗尽的时候c3p0一次同时获取的连接数。默认值: 3
acquireIncrement = 3
# 定义在从数据库获取新连接失败后重复尝试的次数。默认值: 30 ；小于等于0表示无限次
acquireRetryAttempts = 0
# 重新尝试的时间间隔，默认为：1000毫秒
acquireRetryDelay = 1000
# 关闭连接时，是否提交未提交的事务，默认为false，即关闭连接，回滚未提交的事务
autoCommitOnClose = false
# c3p0将建一张名为Test的空表，并使用其自带的查询语句进行测试。如果定义了这个参数那么属性preferredTestQuery将被忽略。你不能在这张Test表上进行任何操作，它将只供c3p0测试使用。默认值: null
automaticTestTable = null
# 如果为false，则获取连接失败将会引起所有等待连接池来获取连接的线程抛出异常，但是数据源仍有效保留，并在下次调用getConnection()的时候继续尝试获取连接。如果设为true，那么在尝试获取连接失败后该数据源将申明已断开并永久关闭。默认: false
breakAfterAcquireFailure = false
# 检查所有连接池中的空闲连接的检查频率。默认值: 0，不检查
idleConnectionTestPeriod = 0
# c3p0全局的PreparedStatements缓存的大小。如果maxStatements与maxStatementsPerConnection均为0，则缓存不生效，只要有一个不为0，则语句的缓存就能生效。如果默认值: 0
maxStatements = 0
# maxStatementsPerConnection定义了连接池内单个连接所拥有的最大缓存statements数。默认值: 0
maxStatementsPerConnection = 0
```

DBCP（不推荐）

```
#==============================================================
# 数据库配置文件样例
```

```
# DsFactory默认读取的配置文件是config/db.setting
# db.setting的配置包括两部分：基本连接信息和连接池配置信息。
# 基本连接信息所有连接池都支持，连接池配置信息根据不同的连接池，连接池配置是根据连接池相应的配置项移植而来
#================================================================================

#--------------------------------------------------------------------------------
## 基本配置信息
# JDBC URL，根据不同的数据库，使用相应的JDBC连接字符串
url = jdbc:mysql://<host>:<port>/<database_name>
# 用户名，此处也可以使用 user 代替
username = 用户名
# 密码，此处也可以使用 pass 代替
password = 密码
# JDBC驱动名，可选（Hutool会自动识别）
driver = com.mysql.jdbc.Driver

## 可选配置
# 是否在日志中显示执行的SQL
showSql = true
# 是否格式化显示的SQL
formatSql = false
# 是否显示SQL参数
showParams = true

#--------------------------------------------------------------------------------
## 连接池配置项

## ------------------------------------------------- Dbcp
# (boolean) 连接池创建的连接的默认的auto-commit 状态
defaultAutoCommit = true
# (boolean) 连接池创建的连接的默认的read-only 状态。 如果没有设置则setReadOnly 方法将不会被调用。（某些驱动不支持只读模式， 比如：Informix)
defaultReadOnly = false
# (String) 连接池创建的连接的默认的TransactionIsolation 状态。 下面列表当中的某一个：( 参考javadoc) NONE READ_COMMITTED EAD_UNCOMMITTED REPEATABLE_READ SERIALIZABLE
defaultTransactionIsolation = NONE
# (int) 初始化连接： 连接池启动时创建的初始化连接数量，1。2 版本后支持
initialSize = 10
# (int) 最大活动连接： 连接池在同一时间能够分配的最大活动连接的数量， 如果设置为非正
```

```
数则表示不限制
maxActive = 100
# (int) 最大空闲连接： 连接池中容许保持空闲状态的最大连接数量， 超过的空闲连接将被释放， 如果设置为负数表示不限制 如果启用，将定期检查限制连接，如果空闲时间超过minEvictableIdleTimeMillis 则释放连接 （ 参考testWhileIdle ）
maxIdle = 8
# (int) 最小空闲连接： 连接池中容许保持空闲状态的最小连接数量， 低于这个数量将创建新的连接， 如果设置为0 则不创建 如果连接验证失败将缩小这个值（ 参考testWhileIdle ）
minIdle = 0
# (int) 最大等待时间： 当没有可用连接时， 连接池等待连接被归还的最大时间( 以毫秒计数)， 超过时间则抛出异常， 如果设置为-1 表示无限等待
maxWait = 30000
# (String) SQL 查询， 用来验证从连接池取出的连接， 在将连接返回给调用者之前。 如果指定， 则查询必须是一个SQL SELECT 并且必须返回至少一行记录 查询不必返回记录，但这样将不能抛出SQL异常
validationQuery = SELECT 1
# (boolean) 指明是否在从池中取出连接前进行检验， 如果检验失败， 则从池中去除连接并尝试取出另一个。注意： 设置为true 后如果要生效，validationQuery 参数必须设置为非空字符串 参考validationInterval以获得更有效的验证
testOnBorrow = false
# (boolean) 指明是否在归还到池中前进行检验 注意： 设置为true 后如果要生效，validationQuery 参数必须设置为非空字符串
testOnReturn = false
# (boolean) 指明连接是否被空闲连接回收器( 如果有) 进行检验。 如果检测失败， 则连接将被从池中去除。注意： 设置为true 后如果要生效，validationQuery 参数必须设置为非空字符串
testWhileIdle = false
```

案例1-导出Blob字段图像

需求：

有一张单表存储着图片（图片使用Blob字段）以及图片的相关信息，需求是从数据库中将这些Blob字段内容保存为图片文件，文件名为图片的相关信息。

环境

数据库：Oracle

本地：Windows

工具：Hutool-db模块

编码

数据库配置：src/main/resources/config/db.setting

```
# ----------------------------------------------------------
# ----- Setting File with UTF8-----
# ----- 数据库配置文件 -----
# ----------------------------------------------------------

#JDBC url，必须
url = jdbc:oracle:thin:@192.168.1.1:1521/orcl
#用户名，必须
user = test
#密码，必须，如果密码为空，请填写 pass =
pass = test
```

代码：PicTransfer.java

```
import java.sql.Blob;
import java.sql.ResultSet;
import java.sql.SQLException;

import com.xiaoleilu.hutool.db.Entity;
import com.xiaoleilu.hutool.db.SqlRunner;
import com.xiaoleilu.hutool.db.ds.DSFactory;
import com.xiaoleilu.hutool.db.handler.RsHandler;
import com.xiaoleilu.hutool.io.FileUtil;
import com.xiaoleilu.hutool.util.CollectionUtil;
import com.xiaoleilu.hutool.util.StrUtil;

public class PicTransfer {
private static SqlRunner runner = SqlRunner.create();
private static String destDir = "f:/pic";

public static void main(String[] args) throws SQLException {
runner.find(
CollectionUtil.newArrayList("NAME", "TYPE", "GROUP", "PIC"),
Entity.create("PIC_INFO").set("TYPE", 1),
new RsHandler<String>(){
@Override
```

```
public String handle(ResultSet rs) throws SQLException {
while(rs.next()){
save(rs);
}
return null;
}
}
);
}

private static void save(ResultSet rs) throws SQLException{
String path = StrUtil.format("{}/{}-{}.jpg", destDir, rs.getString("NAME"),
rs.getString("GROUP"));
FileUtil.writeFromStream(rs.getBlob("PIC").getBinaryStream(), path);
}
}
```

NoSQL

Redis客户端封装-RedisDS

介绍

RedisDS基于Jedis封装，需自行引入Jedis依赖。

使用

引入依赖

```
<dependency>
<groupId>redis.clients</groupId>
<artifactId>jedis</artifactId>
<version>2.9.0</version>
</dependency>
```

配置

在ClassPath（或者src/main/resources）的config目录下下新建redis.setting

```
#----------------------------------------------------------------------
```

```
# Redis客户端配置样例
# 每一个分组代表一个Redis实例
# 无分组的Pool配置为所有分组的共用配置，如果分组自己定义Pool配置，则覆盖共用配置
# 池配置来自于：https://www.cnblogs.com/jklk/p/7095067.html
#--------------------------------------------------------------------------

#----- 默认（公有）配置
# 地址，默认localhost
host = localhost
# 端口，默认6379
port = 6379
# 超时，默认2000
timeout = 2000
# 连接超时，默认timeout
connectionTimeout = 2000
# 读取超时，默认timeout
soTimeout = 2000
# 密码，默认无
password =
# 数据库序号，默认0
database = 0
# 客户端名，默认"Hutool"
clientName = Hutool
# SSL连接，默认false
ssl = false;

#----- 自定义分组的连接
[custom]
# 地址，默认localhost
host = localhost
# 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true
BlockWhenExhausted = true;
# 设置的逐出策略类名, 默认DefaultEvictionPolicy(当连接超过最大空闲时间,或连接数超过最大空闲连接数)
evictionPolicyClassName = org.apache.commons.pool2.impl.DefaultEvictionPolicy
# 是否启用pool的jmx管理功能, 默认true
jmxEnabled = true;
# 是否启用后进先出, 默认true
lifo = true;
# 最大空闲连接数, 默认8个
maxIdle = 8
# 最小空闲连接数, 默认0
```

```
minIdle = 0
# 最大连接数, 默认8个
maxTotal = 8
# 获取连接时的最大等待毫秒数(如果设置为阻塞时BlockWhenExhausted),如果超时就抛异常, 小于零:阻塞不确定的时间,  默认-1
maxWaitMillis = -1
# 逐出连接的最小空闲时间 默认1800000毫秒(30分钟)
minEvictableIdleTimeMillis = 1800000
# 每次逐出检查时 逐出的最大数目 如果为负数就是 : 1/abs(n), 默认3
numTestsPerEvictionRun = 3;
# 对象空闲多久后逐出, 当空闲时间>该值 且 空闲连接>最大空闲数 时直接逐出,不再根据MinEvictableIdleTimeMillis判断  (默认逐出策略)
SoftMinEvictableIdleTimeMillis = 1800000
# 在获取连接的时候检查有效性, 默认false
testOnBorrow = false
# 在空闲时检查有效性, 默认false
testWhileIdle = false
# 逐出扫描的时间间隔(毫秒) 如果为负数,则不运行逐出线程, 默认-1
timeBetweenEvictionRunsMillis = -1
```

构建

```
Jedis jedis = RedisDS.create().getJedis();
```

MongoDB客户端封装-MongoDS

介绍

针对MongoDB客户端封装。客户端需自行引入依赖。

使用

引入依赖

```
<dependency>
<groupId>org.mongodb</groupId>
<artifactId>mongo-java-driver</artifactId>
<version>3.8.1</version>
</dependency>
```

配置

在ClassPath（或者src/main/resources）的config目录下下新建mongo.setting

```
#-------------------------------------
# MongoDB 连接设定
# author xiaoleilu
#-------------------------------------

#每个主机答应的连接数（每个主机的连接池大小），当连接池被用光时，会被阻塞住 ，默以为10 --int
connectionsPerHost=100
#线程队列数，它以connectionsPerHost值相乘的结果就是线程队列最大值。如果连接线程排满了队列就会抛出“Out of semaphores to get db”错误 --int
threadsAllowedToBlockForConnectionMultiplier=10
#被阻塞线程从连接池获取连接的最长等待时间（ms） --int
maxWaitTime = 120000
#在建立（打开）套接字连接时的超时时间（ms），默以为0（无穷） --int
connectTimeout=0
#套接字超时时间;该值会被传递给Socket.setSoTimeout(int)。默以为0（无穷） --int
socketTimeout=0
#是否打开长连接. defaults to false --boolean
socketKeepAlive=false

#------------------------------- MongoDB实例连接
[master]
host = 127.0.0.1:27017

[slave]
host = 127.0.0.1:27018
#-----------------------------------------------------
```

使用

```
//master slave 组成主从集群
MongoDatabase db = MongoFactory.getDS("master", "slave").getDb("test");
```

常见问题

问题描述：no suitable driver found for jdbc

图片地址：https://static.oschina.net/uploads/img/201807/20170806_7Z30.png

解答：出现此类问题一般是JDBC驱动版本不一致导致的，出现此问题的用户使用的是ojdbc14，服务端使用Oracle11g，JDK8，此处升级到ojdbc6即可。

版本对应见：https://www.oracle.com/technetwork/database/application-development/jdbc/downloads/index.html

http客户端(Hutool-http)

概述

由来

在Java的世界中，Http客户端之前一直是Apache家的HttpClient占据主导，但是由于此包较为庞大，API又比较难用，因此并不使用很多场景。而新兴的OkHttp、Jodd-http固然好用，但是面对一些场景时，学习成本还是有一些的。很多时候，我们想追求轻量级的Http客户端，并且追求简单易用。而JDK自带的HttpUrlConnection可以满足大部分需求。Hutool针对此类做了一层封装，使Http请求变得无比简单。

介绍

Hutool-http针对JDK的HttpUrlConnection做一层封装，简化了HTTPS请求、文件上传、Cookie记忆等操作，使Http请求变得无比简单。

Hutool-http的核心集中在两个类：

- HttpRequest
- HttpResponse

同时针对大部分情境，封装了HttpUtil工具类。

Hutool-http优点

1. 根据URL自动判断是请求HTTP还是HTTPS，不需要单独写多余的代码。

2. 表单数据中有File对象时自动转为multipart/form-data表单，不必单做做操作。

3. 默认情况下Cookie自动记录，比如可以实现模拟登录，即第一次访问登录URL后后续请求就是登录状态。

4. 自动识别304跳转并二次请求

5. 自动识别页面编码，即根据header信息或者页面中的相关标签信息自动识别编码，最大可能避免乱码。

6. 自动识别并解压Gzip格式返回内容

使用

最简单的使用莫过于用HttpUtil工具类快速请求某个页面：

```
//GET请求
String content = HttpUtil.get(url);
```

一行代码即可搞定，当然Post请求也很简单：

```
//POST请求
HashMap<String, Object> paramMap = new HashMap<>();
paramMap.put("city", "北京");

String result1 = HttpUtil.post(url, paramMap);
```

Post请求只需使用Map预先制定form表单项即可。

更多

根据Hutool的“便捷性与灵活性并存”原则，HttpUtil的存在体现了便捷性，那HttpRequest对象的使用则体现了灵活性，使用此对象可以自定义更多的属性给请求，以适应Http请求中的不同场景（例如自定义header、自定义cookie、自定义代理等等）。相关类的使用请见下几个章节。

Http客户端工具类-HttpUtil

概述

HttpUtil是应对简单场景下Http请求的工具类封装，这个工具类可以保证在一个方法之内完成Http请求

使用

请求普通页面

针对最为常用的GET和POST请求，HttpUtil封装了两个方法，

- HttpUtil.get
- HttpUtil.post

这两个方法用于请求普通页面，然后返回页面内容的字符串，同时提供一些重载方法用于指定请求参数（指定参数支持File对象，可实现文件上传，当然仅仅针对POST请求）。

GET请求栗子：

```
// 最简单的HTTP请求，可以自动通过header等信息判断编码，不区分HTTP和HTTPS
String result1= HttpUtil.get("https://www.baidu.com");

// 当无法识别页面编码的时候，可以自定义请求页面的编码
String result2= HttpUtil.get("https://www.baidu.com", "UTF-8");
```

```
//可以单独传入http参数，这样参数会自动做URL编码，拼接在URL中
HashMap<String, Object> paramMap = new HashMap<>();
paramMap.put("city", "北京");
String result3= HttpUtil.get("https://www.baidu.com", paramMap);
```

POST请求栗子：

```
HashMap<String, Object> paramMap = new HashMap<>();
paramMap.put("city", "北京");
String result= HttpUtil.post("https://www.baidu.com", paramMap);

//文件上传只需将参数中的键指定（默认file），值设为文件对象即可，对于使用者来说，文件上传与普通表单提交并无区别
paramMap.put("file", FileUtil.file("D:\\face.jpg"));
String result= HttpUtil.post("https://www.baidu.com", paramMap);
```

下载文件

因为Hutool-http机制问题，请求页面返回结果是一次性解析为byte[]的，如果请求URL返回结果太大（比如文件下载），那内存会爆掉，因此针对文件下载HttpUtil单独做了封装。文件下载在面对大文件时采用流的方式读写，内存中只是保留一定量的缓存，然后分块写入硬盘，因此大文件情况下不会对内存有压力。

```
String fileUrl = "http://mirrors.sohu.com/centos/7.3.1611/isos/x86_64/CentOS-7-x86_64-DVD-1611.iso";

//将文件下载后保存在E盘，返回结果为下载文件大小
long size = HttpUtil.downloadFile(fileUrl, FileUtil.file("e:/"));
System.out.println("Download size: " + size);
```

当然，如果我们想感知下载进度，还可以使用另一个重载方法回调感知下载进度：

```
//带进度显示的文件下载
HttpUtil.downloadFile(fileUrl, FileUtil.file("e:/"), new StreamProgress(){

@Override
public void start() {
Console.log("开始下载。。。。");
}

@Override
public void progress(long progressSize) {
Console.log("已下载：{}", FileUtil.readableFileSize(progressSize));
}
```

```
@Override
public void finish() {
Console.log("下载完成！");
}
});
```

StreamProgress接口实现后可以感知下载过程中的各个阶段。

当然，工具类提供了一个更加抽象的方法：HttpUtil.download，此方法会请求URL，将返回内容写入到指定的OutputStream中。使用这个方法，可以更加灵活的将HTTP内容转换写出，以适应更多场景。

更多有用的工具方法

• HttpUtil.encode和HttpUtil.decode 两个方法封装了JDK的URLEncoder.encode和URLDecoder.decode方法，可以方便的对URL参数进行URL编码和解码。

• HttpUtil.toParams和HttpUtil.decodeParams 两个方法是将Map参数转为URL参数字符串和将URL参数字符串转为Map对象

• HttpUtil.urlWithForm是将URL字符串和Map参数拼接为GET请求所用的完整字符串使用

• HttpUtil.getMimeType 根据文件扩展名快速获取其MimeType（参数也可以是完整文件路径）

• HttpUtil.getClientIP 根据指定Http头信息获取客户端IP地址，此方法适用于在Nginx转发时获取真实客户端地址的快捷方法（此方法依赖于Servlet-api）

更多请求参数

如果想设置头信息、超时、代理等信息，请见下一章节《Http客户端-HttpRequest》。

Http客户端-HttpRequest

介绍

本质上，HttpUtil中的get和post工具方法都是HttpRequest对象的封装，因此如果想更加灵活操作Http请求，可以使用HttpRequest。

使用

普通表单

我们以POST请求为例：

```
//链式构建请求
String result2 = HttpRequest.post(url)
```

```
.header(Header.USER_AGENT, "Hutool http")//头信息，多个头信息多次调用此方法即可
.form(paramMap)//表单内容
.timeout(20000)//超时，毫秒
.execute().body();
Console.log(result2);
```

通过链式构建请求，我们可以很方便的指定Http头信息和表单信息，最后调用execute方法即可执行请求，返回HttpResponse对象。HttpResponse包含了服务器响应的一些信息，包括响应的内容和响应的头信息。通过调用body方法即可获取响应内容。

Restful请求

```
String json = ...;
String result2 = HttpRequest.post(url)
.body(json)
.execute().body();
```

其它自定义项

同样，我们通过HttpRequest可以很方便的做以下操作：

- 指定请求头
- 自定义Cookie（cookie方法）
- 指定是否keepAlive（keepAlive方法）
- 指定表单内容（form方法）
- 指定请求内容，比如rest请求指定JSON请求体（body方法）
- 超时设置（timeout方法）
- 指定代理（setProxy方法）
- 指定SSL协议（setSSLProtocol）
- 简单验证（basicAuth方法）

Http响应封装-HttpResponse

介绍

HttpResponse是HttpRequest执行execute()方法后返回的一个对象，我们可以通过此对象获取服务端返回的：

- Http状态码（getStatus方法）
- 返回内容编码（contentEncoding方法）
- 是否Gzip内容（isGzip方法）
- 返回内容（body、bodyBytes、bodyStream方法）

• 响应头信息（header方法）

使用

此对象的使用非常简单，最常用的便是body方法，会返回字符串Http响应内容。如果想获取byte[]则调用bodyBytes即可。

获取响应状态码

```
HttpResoonse res = HttpRequest.post(url)..execute();
Console.log(res.getStatus());
```

获取响应头信息

```
HttpResoonse res = HttpRequest.post(url)..execute();
//预定义的头信息
Console.log(res.header(Header.CONTENT_ENCODING));
//自定义头信息
Console.log(res.header("Content-Disposition"));
```

常用Http状态码-HttpStatus

介绍

针对Http响应，Hutool封装了一个类用于保存Http状态码

此类用于保存一些状态码的别名，例如：

```
/**
* HTTP Status-Code 200: OK.
*/
public static final int HTTP_OK = 200;
```

HTML工具类-HtmlUtil

由来

针对Http请求中返回的Http内容，Hutool使用此工具类来处理一些HTML页面相关的事情。

方法

• HtmlUtil.restoreEscaped 还原被转义的HTML特殊字符

- HtmlUtil.encode 转义文本中的HTML字符为安全的字符
- HtmlUtil.cleanHtmlTag 清除所有HTML标签
- HtmlUtil.removeHtmlTag 清除指定HTML标签和被标签包围的内容
- HtmlUtil.unwrapHtmlTag 清除指定HTML标签，不包括内容
- HtmlUtil.removeHtmlAttr 去除HTML标签中的属性
- HtmlUtil.removeAllHtmlAttr 去除指定标签的所有属性
- HtmlUtil.filter 过滤HTML文本，防止XSS攻击

栗子-爬取开源中国的开源资讯

介绍

为了演示Hutool-http的http请求功能，因此这个栗子用红薯家的开源资讯开刀，在此做个简单的Demo。

开始

分析页面

1. 打开红薯家的主页，我们找到最显眼的开源资讯模块，然后点击“更多”，打开“开源资讯”板块。

图片地址：https://static.oschina.net/uploads/img/201711/19204312_zJD8.png

1. 打开F12调试器，点击快捷键F12打开Chrome的调试器，点击“Network”选项卡，然后在页面上点击“全部资讯”。

图片地址：https://static.oschina.net/uploads/img/201711/19204634_1ahd.png

图片地址：https://static.oschina.net/uploads/img/201711/19204743_eJBy.png

1. 由于红薯家的列表页是通过下拉翻页的，因此下拉到底部会触发第二页的加载，此时我们下拉到底部，然后观察调试器中是否有新的请求出现。如图，我们发现第二个请求是列表页的第二页。

图片地址：https://static.oschina.net/uploads/img/201711/19205000_V7Sj.png

1. 我们打开这个请求地址，可以看到纯纯的内容。红框所指地址为第二页的内容，很明显p参数代表了页码page。

图片地址：https://static.oschina.net/uploads/img/201711/19205156_dTb8.png

1. 我们右键点击后查看源码，可以看到源码。

图片地址：https://static.oschina.net/uploads/img/201711/19205356_lQBd.png

1. 找到标题部门的HTML源码，然后搜索这个包围这个标题的HTML部分，看是否可以定位标题。

图片地址：https://static.oschina.net/uploads/img/201711/19205448_Cn4w.png

图片地址：https://static.oschina.net/uploads/img/201711/19205601_HgAv.png

至此分析完毕，我们拿到了列表页的地址，也拿到了可以定位标题的相关字符（在后面用正则提取标题用），就可以开始使用Hutool编码了。

模拟Http请求爬取页面

使用Hutool-http配合ReUtil请求并提取页面内容非常简单，代码如下：

```
//请求列表页
String listContent =
HttpUtil.get("https://www.oschina.net/action/ajax/get_more_news_list?newsType=&p=
2");
//使用正则获取所有标题
List<String> titles = ReUtil.findAll("<span class=\"text-ellipsis\">(.*?)</span>",
listContent, 1);
for (String title : titles) {
//打印标题
Console.log(title);
}
```

抓取结果为：

图片地址：https://static.oschina.net/uploads/img/201711/19210437_aJDc.png

其实核心就前两行代码，第一行请求页面内容，第二行正则定位所有标题行并提取标题部分。

这里我解释下正则部分：ReUtil.findAll方法用于查找所有匹配正则表达式的内容部分，第二个参数1表示提取第一个括号（分组）中的内容，0表示提取所有正则匹配到的内容。这个方法可以看下core模块中ReUtil章节了解详情。

(.*?)这个正则就是我们上面分析页面源码后得到的正则，其中`(.?)`*表示我们需要的内容*，`.`*表示任意字符*，`*`表示0个或多个，`?`表示最短匹配，整个正则的意思就是。，以`<span class="text-ellipsis">`开头，`</span>`结尾的中间所有字符，中间的字符要达到最短。`?`的作用其实就是将范围限制到最小，不然`</span>`很可能匹配到后面去了。

关于正则表达式这块可以看下我的博客：正则表达式简明参考(http://luxiaolei.com/regex-guide)

结语

不得不说，抓取本身并不困难，尤其配合Hutool会让这项工作变得更加简单快速，而其中的难点便是分析页面和定位我们需要的内容。

真正的内容抓取分为连个部分：

- 找到列表页（很多网站都没有一个总的列表页）
- 请求列表页，获取详情页地址
- 请求详情页并使用正则匹配我们需要的内容
- 入库或将内容保存为文件

而且在抓取过程中我们也会遇到各种问题，包括但不限于：

- 封IP

• 对请求Header有特殊要求

• 对Cookie有特殊要求

• 验证码

这些问题都有一些解决办法，具体要在具体的开发中分析解决。

希望大家看到这个栗子有所启发，也为Hutool提供更多更好的意见~

常见问题

Received fatal alert: handshake_failure 错误

图片地址：https://static.oschina.net/uploads/img/201807/22184830_EHJ0.png

用户错误如图，场景为使用Hutool-http请求https服务器，原因是JDK中的JCE安全机制导致的问题解决方法如下：

• 方法1：如果你使用的是JDK8，请升级到JDK8的最新版本（例如jdk1.8.0_181）。

• 方法2：尝试添加以下代码：

```
System.setProperty("https.protocols", "TLSv1.2,TLSv1.1,SSLv3");
```

定时任务(Hutool-cron)

定时任务模块概述

由来

Java中定时任务使用的最多的我想就是quartz(http://www.quartz-scheduler.org/)了，但是这个框架太过庞大，而且我也不需要用到这么多东西，使用方法也是比较复杂（官方Demo我实在是无语……）。

用过Linux的crontab的人都知道，使用其定时的表达式可以非常灵活的定义定时任务的时间以及频率（Linux的crontab精确到分，而Quartz的精确到秒，不过对我来说精确到分已经够用了，精确到秒的可以使用Timer可以搞定），然后就是crontab的那个迷人的配置文件，可以把定时任务很清晰的罗列出来，这个我也是比较喜欢的。（记得当时Spring整合Quartz的时候那XML看的我眼都花了……）。于是Hutool-cron诞生。

介绍

Hutool的定时任务模块与Linux的Crontab使用上非常类似，通过一个cron.setting配置文件，简单调用start()方法即可简单使用。

同时还提供了秒匹配和年匹配等Quartz才有的功能，定时任务表达式上也同时兼容Crontab（Cron4j）和Quartz的表达式。

全局定时任务-CronUtil

介绍

CronUtil通过一个全局的定时任务配置文件，实现统一的定时任务调度。

使用

1、配置文件

对于Maven项目，首先在src/main/resources/config下放入cron.setting文件（默认是这个路径的这个文件），然后在文件中放入定时规则，规则如下：

```
# 我是注释
[com.company.aaa.job]
TestJob.run = */10 * * * *
TestJob2.run = */10 * * * *
```

中括号表示分组，也表示需要执行的类或对象方法所在包的名字，这种写法有利于区分不同业务的定时任务。

TestJob.run表示需要执行的类名和方法名（通过反射调用），*/10 *表示定时任务表达式，此处表示每10分钟执行一次，以上配置等同于：

```
com.company.aaa.job.TestJob.run = */10 * * * *
com.company.aaa.job.TestJob2.run = */10 * * * *
```

> 提示
>
> 关于表达式语法，见：http://www.cnblogs.com/peida/archive/2013/01/08/2850483.html(http://www.cnblogs.com/peida/archive/2013/01/08/2850483.html)

2、启动

```
CronUtil.start();
```

如果想让执行的作业同定时任务线程同时结束，可以将定时任务设为守护线程，需要注意的是，此模式下会在调用stop时立即结束所有作业线程，请确保你的作业可以被中断：

```
//使用deamon模式，
CronUtil.start(true);
```

3、关闭

```
CronUtil.stop();
```

更多选项

秒匹配和年匹配

考虑到Quartz表达式的兼容性，且存在对于秒级别精度匹配的需求，Hutool可以通过设置使用秒匹配模式来兼容。

```
//支持秒级别定时任务
CronUtil.setMatchSecond(true);
```

此时Hutool可以兼容Quartz表达式（5位表达式、6位表达式都兼容）

动态添加定时任务

当然，如果你想动态的添加定时任务，使用CronUtil.schedule(String schedulingPattern, Runnable task)方法即可（使用此方法加入的定时任务不会被写入到配置文件）。

```
CronUtil.schedule("*/2 * * * * *", new Task() {
@Override
public void execute() {
Console.log("Task excuted.");
}
});

// 支持秒级别定时任务
CronUtil.setMatchSecond(true);
CronUtil.start();
```

扩展(Hutool-extra)

概述

由来

由于Hutool的原则是不依赖于其它配置文件，但是很多时候我们需要针对第三方非常棒的库做一些工具类化的支持，因此Hutoo-extra包主要用于支持第三方库的工具类支持。

介绍

现阶段扩展包括：

模板引擎封装工具类

Servlet封装

Jsch库封装（SSH和Sftp）

Apache Commons Net封装（FTP部分）

邮件封装

Zxing封装（二维码）

邮件工具-MailUtil

概述

在Java中发送邮件主要品依靠javax.mail包，但是由于使用比较繁琐，因此Hutool针对其做了封装。由于依赖第三方包，因此将此工具类归类到extra模块中。

使用

引入依赖

Hutool对所有第三方都是可选依赖，因此在使用MailUtil时需要自行引入第三方依赖。

```
<dependency>
<groupId>javax.mail</groupId>
<artifactId>mail</artifactId>
<version>1.4.7</version>
</dependency>
```

邮件服务器配置

在classpath（在标准Maven项目中为src/main/resources）的config目录下新建mail.setting文件，最小配置内容如下，在此配置下，smtp服务器和用户名都将通过from参数识别：

```
# 发件人（必须正确，否则发送失败）
from = hutool@yeah.net
# 密码（注意，某些邮箱需要为SMTP服务单独设置密码，详情查看相关帮助）
```

```
pass = q1w2e3
```

有时候一些非标准邮箱服务器（例如企业邮箱服务器）的smtp地址等信息并不与发件人后缀一致，端口也可能不同，此时Hutool可以提供完整的配置文件：

完整配置

```
# 邮件服务器的SMTP地址，可选，默认为smtp.<发件人邮箱后缀>
host = smtp.yeah.net
# 邮件服务器的SMTP端口，可选，默认25
port = 25
# 发件人（必须正确，否则发送失败）
from = hutool@yeah.net
# 用户名，默认为发件人邮箱前缀
user = hutool
# 密码（注意，某些邮箱需要为SMTP服务单独设置密码，详情查看相关帮助）
pass = q1w2e3
```

> 注意
>
> 邮件服务器必须支持并打开SMTP协议，详细请查看相关帮助说明
>
> 配置文件的样例中提供的是我专门为测试邮件功能注册的yeah.net邮箱，帐号密码公开，供Hutool用户测试使用。

发送邮件

1. 发送普通文本邮件，最后一个参数可选是否添加多个附件：

```
MailUtil.send("hutool@foxmail.com", "测试", "邮件来自Hutool测试", false);
```

1. 发送HTML格式的邮件并附带附件，最后一个参数可选是否添加多个附件：

```
MailUtil.send("hutool@foxmail.com", "测试", "<h1>邮件来自Hutool测试</h1>", true,
FileUtil.file("d:/aaa.xml"));
```

1. 群发邮件，可选HTML或普通文本，可选多个附件：

```
ArrayList<String> tos = CollUtil.newArrayList(
"person1@bbb.com",
"person2@bbb.com",
"person3@bbb.com",
"person4@bbb.com");

MailUtil.send(tos, "测试", "邮件来自Hutool群发测试", false);
```

发送邮件非常简单，只需一个方法即可搞定其中按照参数顺序说明如下：

1. tos:　对方的邮箱地址，可以是单个，也可以是多个（Collection表示）

2. subject：标题

3. content：邮件正文，可以是文本，也可以是HTML内容

4. isHtml： 是否为HTML，如果是，那参数3识别为HTML内容

5. files： 可选：附件，可以为多个或没有，将File对象加在最后一个可变参数中即可

其它

1. 自定义邮件服务器

除了使用配置文件定义全局账号以外，MailUtil.send方法同时提供重载方法可以传入一个MailAccount对象，这个对象为一个普通Bean，记录了邮件服务器信息。

```
MailAccount account = new MailAccount();
account.setHost("smtp.yeah.net");
account.setPort("25");
account.setAuth(true);
account.setFrom("hutool@yeah.net");
account.setUser("hutool");
account.setPass("q1w2e3");

MailUtil.send(account, CollUtil.newArrayList("hutool@foxmail.com"), "测试", "邮件来自Hutool测试", false);
```

1. 使用SSL加密方式发送邮件

在使用QQ或Gmail邮箱时，需要强制开启SSL支持，此时我们只需修改配置文件即可：

```
# 发件人（必须正确，否则发送失败），“小磊”可以任意变更，<>内的地址必须唯一，以下方式也对
# from = hutool@yeah.net
from = 小磊<hutool@yeah.net>
# 密码（注意，某些邮箱需要为SMTP服务单独设置密码，详情查看相关帮助）
pass = q1w2e3
# 使用SSL安全连接
sslEnable = true
```

在原先极简配置下只需加入sslEnable即可完成SSL连接，当然，这是最简单的配置，很多参数根据已有参数已设置为默认。

完整的配置文件如下：

```
# 邮件服务器的SMTP地址
host = smtp.yeah.net
# 邮件服务器的SMTP端口
port = 465
# 发件人（必须正确，否则发送失败）
```

```
from = hutool@yeah.net
# 用户名（注意：如果使用foxmail邮箱，此处user为qq号）
user = hutool
# 密码（注意，某些邮箱需要为SMTP服务单独设置密码，详情查看相关帮助）
pass = q1w2e3
#使用 STARTTLS安全连接，STARTTLS是对纯文本通信协议的扩展。
startttlsEnable = true

# 使用SSL安全连接
sslEnable = true
# 指定实现javax.net.SocketFactory接口的类的名称,这个类将被用于创建SMTP的套接字
socketFactoryClass = javax.net.ssl.SSLSocketFactory
# 如果设置为true,未能创建一个套接字使用指定的套接字工厂类将导致使用java.net.Socket创建的套接字类, 默认值为true
socketFactoryFallback = true
# 指定的端口连接到在使用指定的套接字工厂。如果没有设置,将使用默认端口456
socketFactoryPort = 465

# SMTP超时时长，单位毫秒，缺省值不超时
timeout = 0
# Socket连接超时值，单位毫秒，缺省值不超时
connectionTimeout = 0
```

3、针对QQ邮箱和Foxmail邮箱的说明

(1) QQ邮箱中SMTP密码是单独生成的授权码，而非你的QQ密码，至于怎么生成，见腾讯的帮助说明：http://service.mail.qq.com/cgi-bin/help?subtype=1&&id=28&&no=1001256(http://service.mail.qq.com/cgi-bin/help?subtype=1&&id=28&&no=1001256)

使用帮助引导生成授权码后，配置如下即可：

```
pass = 你生成的授权码
```

(2) Foxmail邮箱本质上也是QQ邮箱的一种别名，你可以在你的QQ邮箱中设置一个foxmail邮箱，不过配置上有所区别。在Hutool中user属性默认提取你邮箱@前面的部分，但是foxmail邮箱是无效的，需要单独配置为与之绑定的qq号码或者XXXX@qq.com的XXXX。即：

```
host = smtp.qq.com
from = XXXX@foxmail.com
user = foxmail邮箱对应的QQ号码或者qq邮箱@前面部分
...
```

Servlet工具-ServletUtil

由来

最早Servlet相关的工具并不在Hutool的封装考虑范围内，但是后来很多人提出需要一个Servlet Cookie工具，于是我决定建立ServletUtil，这样工具的使用范围就不仅限于Cookie，还包括参数等等。

其实最早的Servlet封装来自于作者的一个MVC框架：Hulu(https://gitee.com/loolly/hulu)，这个MVC框架对Servlet做了一层封装，使请求处理更加便捷。于是Hutool将Hulu中Request类和Response类中的方法封装于此。

使用

加入依赖

```
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>javax.servlet-api</artifactId>
<version>3.1.0</version>
<!-- 此包一般在Servlet容器中都有提供 -->
<scope>provided</scope>
</dependency>
```

方法

- getParamMap 获得所有请求参数
- fillBean 将请求参数转为Bean
- getClientIP 获取客户端IP，支持从Nginx头部信息获取，也可以自定义头部信息获取位置
- getHeader、getHeaderIgnoreCase 获得请求header中的信息
- isIE 客户浏览器是否为IE
- isMultipart 是否为Multipart类型表单，此类型表单用于文件上传
- getCookie 获得指定的Cookie
- readCookieMap 将cookie封装到Map里面
- addCookie 设定返回给客户端的Cookie
- write 返回数据给客户端
- setHeader 设置响应的Header

二维码工具-QrCodeUtil

由来

由于大家对二维码的需求较多，对于二维码的生成和解析我认为应该作为简单的工具存在于Hutool中。考虑到自行实现的难度，因此Hutool针对被广泛接受的的zxing(https://github.com/zxing/zxing)库进行封装。而由于涉及第三方包，因此归类到extra模块中。

使用

引入zxing

考虑到Hutool的非强制依赖性，因此zxing需要用户自行引入：

```
<dependency>
<groupId>com.google.zxing</groupId>
<artifactId>core</artifactId>
<version>3.3.1</version>
</dependency>
```

> 说明
>
> zxing-3.3.1是此文档编写时的最新版本，理论上你引入的版本应比这个版本新。

生成二维码

在此我们将Hutool主页的url生成为二维码，微信扫一扫可以看到H5主页哦：

```
// 生成指定url对应的二维码到文件，宽和高都是300像素
QrCodeUtil.generate("http://hutool.cn/", 300, 300, FileUtil.file("d:/qrcode.jpg"));
```

效果qrcode.jpg：

图片地址：https://static.oschina.net/uploads/img/201801/23203646_3TUp.jpg

自定义参数（since 4.1.2）

通过QrConfig配置对象可以自定义二维码的更对参数，例如长、宽、二维码的颜色、背景颜色、边距等参数，使用方法如下：

```
QrConfig config = new QrConfig(300, 300);
// 设置边距，既二维码和背景之间的边距
config.setMargin(3);
// 设置前景色，既二维码颜色（青色）
config.setForeColor(Color.CYAN.getRGB());
// 设置背景色（灰色）
config.setBackColor(Color.GRAY.getRGB());
// 生成二维码到文件，也可以到流
```

```
QrCodeUtil.generate("http://hutool.cn/", config, FileUtil.file("e:/qrcode.jpg"));
```

效果qrcode.jpg:

图片地址：https://static.oschina.net/uploads/img/201807/15113057_Zc8G.jpg

识别二维码

```
// decode -> "http://hutool.cn/"
String decode = QrCodeUtil.decode(FileUtil.file("d:/qrcode.jpg"));
```

模板引擎

模板引擎封装-TemplateUtil

介绍

随着前后分离的流行，JSP技术和模板引擎慢慢变得不再那么重要，但是早某些场景中（例如邮件模板、页面静态化等）依旧无可可替代，但是各种模板引擎语法大相径庭，使用方式也不尽相同，学习成本很高。Hutool旨在封装各个引擎的共性，使用户只关注模板语法即可，减少学习成本。

Hutool现在封装的引擎有：

- Beetl(http://ibeetl.com/)
- Enjoy(https://gitee.com/jfinal/enjoy)
- Rythm(http://rythmengine.org/)
- FreeMarker(https://freemarker.apache.org/)
- Velocity(http://velocity.apache.org/)

原理

类似于Java日志门面的思想，Hutool将模板引擎的渲染抽象为两个概念：

- Engine 模板引擎，用于封装模板对象，配置各种配置
- Template 模板对象，用于配合参数渲染产生内容

通过实现这两个接口，用户便可抛开模板实现，从而渲染模板。Hutool同时会通过EngineFactory ***根据用户引入的模板引擎库的jar来自动选择用哪个引擎来渲染。**

使用

从字符串模板渲染内容

```
//自动根据用户引入的模板引擎库的jar来自动选择使用的引擎
//TemplateConfig为模板引擎的选项，可选内容有字符编码、模板路径、模板加载方式等，默认通过模板字符串渲染
Engine engine = TemplateUtil.createEngine(new TemplateConfig());

//假设我们引入的是Beetl引擎，则：
Template template = engine.getTemplate("Hello ${name}");
//Dict本质上为Map，此处可用Map
String result = template.render(Dict.create().set("name", "Hutool"));
//输出：Hello Hutool
```

也就是说，使用Hutool之后，无论你用任何一种模板引擎，代码不变（只变更模板内容）。

从classpath查找模板渲染

只需修改TemplateConfig配置文件内容即可更换（这里以Velocity为例）：

```
Engine engine = TemplateUtil.createEngine(new TemplateConfig("templates", ResourceMode.CLASSPATH));
Template template = engine.getTemplate("templates/velocity_test.vtl");
String result = template.render(Dict.create().set("name", "Hutool"));
```

其它方式查找模板

查找模板的方式由ResourceMode定义，包括：

- CLASSPATH 从ClassPath加载模板
- FILE 从File本地目录加载模板
- WEB_ROOT 从WebRoot目录加载模板
- STRING 从模板文本加载模板
- COMPOSITE 复合加载模板（分别从File、ClassPath、Web-root、String方式尝试查找模板）

Jsch封装

Jsch(SSH)工具-JschUtil

由来

此工具最早来自于我的早期项目：Common-tools，当时是为了解决在存在堡垒机（跳板机）环境时无法穿透堡垒机访问内部主机端口问题，于是辗转找到了jsch(http://www.jcraft.com/jsch/)库。为了更加便捷的、且容易理解的方式使用此库，因此有了JschUtil。

使用

引入jsch

```
<dependency>
<groupId>com.jcraft</groupId>
<artifactId>jsch</artifactId>
<version>0.1.54</version>
</dependency>
```

> 说明
>
> 截止本文档撰写完毕，jsch的最新版为0.1.54，理论上应引入的版本应大于或等于此版本。

使用

ssh连接到远程主机

```
//新建会话，此会话用于ssh连接到跳板机（堡垒机），此处为10.1.1.1:22
Session session = JschUtil.getSession("10.1.1.1", 22, "test", "123456");
```

端口映射

```
//新建会话，此会话用于ssh连接到跳板机（堡垒机），此处为10.1.1.1:22
Session session = JschUtil.getSession("10.1.1.1", 22, "test", "123456");

// 将堡垒机保护的内网8080端口映射到localhost，我们就可以通过访问
http://localhost:8080/访问内网服务了
JschUtil.bindPort(session, "172.20.12.123", 8080, 8080);
```

其它方法

- generateLocalPort 生成一个本地端口（从10001开始尝试，找到一个未被使用的本地端口）
- unBindPort 解绑端口映射
- openAndBindPortToLocal 快捷方法，将连接到跳板机和绑定远程主机端口到本地使用一个方法搞定
- close 关闭SSH会话

SFTP封装-Sftp

介绍

SFTP是Secure File Transfer Protocol的缩写，安全文件传送协议。可以为传输文件提供一种安全的加密方法。

SFTP 为 SSH的一部份，是一种传输文件到服务器的安全方式。SFTP是使用加密传输认证信息和传输的数据，所以，使用SFTP是非常安全的。

但是，由于这种传输方式使用了加密/解密技术，所以传输效率比普通的FTP要低得多，如果您对网络安全性要求更高时，可以使用SFTP代替FTP。

使用

引入jsch

```
<dependency>
<groupId>com.jcraft</groupId>
<artifactId>jsch</artifactId>
<version>0.1.54</version>
</dependency>
```

使用

```
Sftp sftp= JschUtil.createSftp("172.0.0.1", 22, "root", "123456");
//进入远程目录
sftp.cd("/opt/upload");
//上传本地文件
sftp.put("e:/test.jpg", "/opt/upload");
//下载远程文件
sftp.get("/opt/upload/test.jpg", "e:/test2.jpg");

//关闭连接
sftp.close();
```

CommonsNet封装

FTP封装-Ftp

介绍

FTP客户端封装，此客户端基于Apache Commons Net(http://commons.apache.org/proper/commons-net/)。

使用

引入依赖

```
<dependency>
<groupId>commons-net</groupId>
<artifactId>commons-net</artifactId>
<version>3.6</version>
</dependency>
```

使用

```
//匿名登录（无需帐号密码的FTP服务器）
Ftp ftp = new Ftp("172.0.0.1");
//进入远程目录
ftp.cd("/opt/upload");
//上传本地文件
ftp.upload("/opt/upload", FileUtil.file("e:/test.jpg"));
//下载远程文件
ftp.download("/opt/upload", "test.jpg", FileUtil.file("e:/test2.jpg"));

//关闭连接
ftp.close();
```

布隆过滤(Hutool-bloomFilter)

概述

切面(Hutool-aop)

概述

Hutool-aop概述

AOP模块主要针对JDK中动态代理进行封装，抽象动态代理为切面类Aspect，通过ProxyUtil代理工具类将切面对象与被代理对象融合，产生一个代理对象，从而可以针对每个方法执行前后做通用的功能。

在aop模块中，默认实现可以下两个切面对象：

1. SimpleAspect 简单切面对象，不做任何操作，继承此对象可重写需要的方法即可，不必实现所

有方法

2. TimeIntervalAspect 执行时间切面对象，用于简单计算方法执行时间，然后通过日志打印方法执行时间

> 由于AOP模块封装JDK的代理，故被代理对象必须实现接口。

切面代理工具-ProxyUtil

使用

使用JDK的动态代理实现切面

1. 我们定义一个接接口：

```
public interface Animal{
void eat();
}
```

1. 定义一个实现类：

```
public class Cat implements Animal{

@Override
public void eat() {
Console.log("猫吃鱼");
}

}
```

1. 我们使用TimeIntervalAspect这个切面代理上述对象，来统计猫吃鱼的执行时间：

```
Animal cat = ProxyUtil.proxy(new Cat(), TimeIntervalAspect.class);
cat.eat();
```

TimeIntervalAspect位于cn.hutool.aop.aspects包，继承自SimpleAspect，代码如下：

```
public class TimeIntervalAspect extends SimpleAspect{
//TimeInterval为Hutool实现的一个计时器
private TimeInterval interval = new TimeInterval();

@Override
public boolean before(Object target, Method method, Object[] args) {
interval.start();
return true;
```

```
}

@Override
public boolean after(Object target, Method method, Object[] args) {
Console.log("Method [{}.{}] execute spend [{}]ms", target.getClass().getName(),
method.getName(), interval.intervalMs());
return true;
}
}
```

执行结果为：

```
猫吃鱼
Method [cn.hutool.aop.test.AopTest$Cat.eat] execute spend [16]ms
```

> 在调用proxy方法后，IDE自动补全返回对象为Cat，因为JDK机制的原因，我们的返回值必须是被代理类实现的接口，因此需要手动将返回值改为**Animal**，否则会报类型转换失败。

使用Cglib实现切面

使用Cglib的好处是无需定义接口即可对对象直接实现切面，使用方式完全一致：

1. 引入Cglib依赖

```
<dependency>
<groupId>cglib</groupId>
<artifactId>cglib</artifactId>
<version>3.2.7</version>
</dependency>
```

1. 定义一个无接口类（此类有无接口都可以）

```
public class Dog {
public String eat() {
Console.log("狗吃肉");
}
}
```

```
Dog dog = ProxyUtil.proxy(new Dog(), TimeIntervalAspect.class);
String result = dog.eat();
```

执行结果为：

```
狗吃肉
Method [cn.hutool.aop.test.AopTest$Dog.eat] execute spend [13]ms
```

其它方法

ProxyUtil中还提供了一些便捷的Proxy方法封装，例如newProxyInstance封装了Proxy.newProxyInstance方法，提供泛型返回值，并提供更多参数类型支持。

原理

动态代理对象的创建原理是假设创建的代理对象名为 $Proxy0：

1. 根据传入的interfaces动态生成一个类，实现interfaces中的接口

2. 通过传入的classloder将刚生成的类加载到jvm中。即将$Proxy0类load

3. 调用$Proxy0的$Proxy0(InvocationHandler)构造函数 创建$Proxy0的对象，并且用interfaces参数遍历其所有接口的方法，并生成实现方法，这些实现方法的实现本质上是通过反射调用被代理对象的方法。

4. 将$Proxy0的实例返回给客户端。

5. 当调用代理类的相应方法时，相当于调用 InvocationHandler.invoke(Object, Method, Object[]) 方法。

脚本(Hutool-script)

概述

介绍

script模块主要针对Java的javax.script封装，可以运行Javascript脚本。

此模块非常简单。主要功能集中在SCriptUtil这个工具类中。

Script工具-ScriptUtil

介绍

针对Script执行工具化封装

使用

1. ScriptUtil.eval 执行Javascript脚本，参数为脚本字符串。

栗子：

```
ScriptUtil.eval("print('Script test!');");
```

1. ScriptUtil.compile 编译脚本，返回一个CompiledScript对象

栗子：

```
CompiledScript script = ScriptUtil.compile("print('Script test!');");
try {
script.eval();
} catch (ScriptException e) {
throw new ScriptRuntimeException(e);
}
```

Office文档操作（Hutool-poi）

概述

由来

Java针对MS Office的操作的库屈指可数，比较有名的就是Apache的POI库。这个库异常强大，但是使用起来也并不容易。Hutool针对POI封装一些常用工具，使Java操作Excel等文件变得异常简单。

介绍

Hutool-poi是针对Apache POI的封装，因此需要用户自行引入POI库,Hutool默认不引入。到目前为止，Hutool-poi支持：

- Excel文件（xls, xlsx）的读取（ExcelReader）
- Excel文件（xls，xlsx）的写出（ExcelWriter）

使用

引入POI依赖

推荐引入poi-ooxml，这个包会自动关联引入poi包，且可以很好的支持Office2007+的文档格式

```
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>${poi.version}</version>
</dependency>
```

如果需要使用Sax方式读取Excel，需要引入以下依赖：

```
<dependency>
    <groupId>xerces</groupId>
    <artifactId>xercesImpl</artifactId>
```

```
    <version>${xerces.version}</version>
</dependency>
```

> 说明
>
> poi-ooxml 版本需高于 3.17（别问我3.8版本为啥不行，因为3.17 > 3.8 ）
>
> xercesImpl版本高于2.11.0

引入后即可使用Hutool的方法操作Office文件了，Hutool提供的类有：

- ExcelUtil Excel工具类，读取的快捷方法都被封装于此
- ExcelReader Excel读取器，Excel读取的封装，可以直接构造后使用。
- ExcelWriter Excel生成并写出器，Excel写出的封装（写出到流或者文件），可以直接构造后使用。

ExcelUtil

介绍

Excel操作工具封装

使用

1. 从文件中读取Excel为ExcelReader

```
ExcelReader reader = ExcelUtil.getReader(FileUtil.file("test.xlsx"));
```

1. 从流中读取Excel为ExcelReader（比如从ClassPath中读取Excel文件）

```
ExcelReader reader = ExcelUtil.getReader(ResourceUtil.getStream("aaa.xlsx"));
```

1. 读取指定的sheet

```
ExcelReader reader;

//通过sheet编号获取
reader = ExcelUtil.getReader(FileUtil.file("test.xlsx"), 0);
//通过sheet名获取
reader = ExcelUtil.getReader(FileUtil.file("test.xlsx"), "sheet1");
```

1. 读取大数据量的Excel

```
private RowHandler createRowHandler() {
return new RowHandler() {
@Override
public void handle(int sheetIndex, int rowIndex, List<Object> rowlist) {
```

```
Console.log("[{}] [{}] {}", sheetIndex, rowIndex, rowlist);
}
};
}

ExcelUtil.readBySax("aaa.xlsx", 0, createRowHandler());
```

后续

ExcelUtil.getReader方法只是将实体Excel文件转换为ExcelReader对象进行操作。接下来请参阅章节ExcelReader对Excel工作簿进行具体操作。

Excel读取-ExcelReader

介绍

读取Excel内容的封装，通过构造ExcelReader对象，指定被读取的Excel文件、流或工作簿，然后调用readXXX方法读取内容为指定格式。

使用

1. 读取Excel中所有行和列，都用列表表示

```
ExcelReader reader = ExcelUtil.getReader("d:/aaa.xlsx");
List<List<Object>> readAll = reader.read();
```

1. 读取为Map列表，默认第一行为标题行，Map中的key为标题，value为标题对应的单元格值。

```
ExcelReader reader = ExcelUtil.getReader("d:/aaa.xlsx");
List<Map<String,Object>> readAll = reader.readAll();
```

1. 读取为Bean列表，Bean中的字段名为标题，字段值为标题对应的单元格值。

```
ExcelReader reader = ExcelUtil.getReader("d:/aaa.xlsx");
List<Person> all = reader.readAll(Person.class);
```

流方式读取Excel2007-Excel07SaxReader

介绍

在标准的ExcelReader中，如果数据量较大，读取Excel会非常缓慢，并有可能造成内存溢出。因此针对大数据量的Excel，Hutool封装了Sax模式的读取方式。

Excel07SaxReader只支持Excel2007格式的Sax读取。

使用

定义行处理器

首先我们实现一下RowHandler接口，这个接口是Sax读取的核心，通过实现handle方法编写我们要对每行数据的操作方式（比如按照行入库，入List或者写出到文件等），在此我们只是在控制台打印。

```
private RowHandler createRowHandler() {
return new RowHandler() {
@Override
public void handle(int sheetIndex, int rowIndex, List<Object> rowlist) {
Console.log("[{}] [{}] {}", sheetIndex, rowIndex, rowlist);
}
};
}
```

ExcelUtil快速读取

```
ExcelUtil.read07BySax("aaa.xlsx", 0, createRowHandler());
```

构建对象读取

```
Excel07SaxReader reader = new Excel07SaxReader(createRowHandler());
reader.read("d:/text.xlsx", 0);
```

reader方法的第二个参数是sheet的序号，-1表示读取所有sheet，0表示第一个sheet，依此类推。

流方式读取Excel2003-Excel03SaxReader

介绍

在标准的ExcelReader中，如果数据量较大，读取Excel会非常缓慢，并有可能造成内存溢出。因此针对大数据量的Excel，Hutool封装了event模式的读取方式。

Excel03SaxReader只支持Excel2003格式的Sax读取。

使用

定义行处理器

首先我们实现一下RowHandler接口，这个接口是Sax读取的核心，通过实现handle方法编写我们要对每行数据的操作方式（比如按照行入库，入List或者写出到文件等），在此我们只是在控制台

打印。

```
private RowHandler createRowHandler() {
return new RowHandler() {
@Override
public void handle(int sheetIndex, int rowIndex, List<Object> rowlist) {
Console.log("[{}] [{}] {}", sheetIndex, rowIndex, rowlist);
}
};
}
```

ExcelUtil快速读取

```
ExcelUtil.read03BySax("aaa.xls", 1, createRowHandler());
```

构建对象读取

```
Excel07SaxReader reader = new Excel07SaxReader(createRowHandler());
reader.read("aaa.xlsx", 0);
```

reader方法的第二个参数是sheet的序号，-1表示读取所有sheet，0表示第一个sheet，依此类推。

Excel生成-ExcelWriter

由来

Excel有读取也便有写出，Hutool针对将数据写出到Excel做了封装。

原理

Hutool将Excel写出封装为ExcelWriter，原理为包装了Workbook对象，每次调用merge（合并单元格）或者write（写出数据）方法后只是将数据写入到Workbook，并不写出文件，只有调用flush或者close方法后才会真正写出文件。

由于机制原因，在写出结束后需要关闭ExcelWriter对象，调用close方法即可关闭，此时才会释放Workbook对象资源，否则带有数据的Workbook一直会常驻内存。

使用栗子

1. 将行列对象写出到Excel

我们先定义一个嵌套的List，List的元素也是一个List，内层的一个List代表一行数据，每行都有4个单元格，最终list对象代表多行数据。

```
List<String> row1 = CollUtil.newArrayList("aa", "bb", "cc", "dd");
List<String> row2 = CollUtil.newArrayList("aa1", "bb1", "cc1", "dd1");
List<String> row3 = CollUtil.newArrayList("aa2", "bb2", "cc2", "dd2");
List<String> row4 = CollUtil.newArrayList("aa3", "bb3", "cc3", "dd3");
List<String> row5 = CollUtil.newArrayList("aa4", "bb4", "cc4", "dd4");

List<List<String>> rows = CollUtil.newArrayList(row1, row2, row3, row4, row5);
```

然后我们创建ExcelWriter对象后写出数据：

```
//通过工具类创建writer
ExcelWriter writer = ExcelUtil.getWriter("d:/writeTest.xlsx");
//通过构造方法创建writer
//ExcelWriter writer = new ExcelWriter("d:/writeTest.xls");

//跳过当前行，既第一行，非必须，在此演示用
writer.passCurrentRow();

//合并单元格后的标题行，使用默认标题样式
writer.merge(list1.size() - 1, "测试标题");
//一次性写出内容
writer.write(rows);
//关闭writer，释放内存
writer.close();
```

效果：

图片地址：https://static.oschina.net/uploads/img/201711/12111543_dmjs.png

2. 写出Map数据

构造数据：

```
Map<String, Object> row1 = new LinkedHashMap<>();
row1.put("姓名", "张三");
row1.put("年龄", 23);
row1.put("成绩", 88.32);
row1.put("是否合格", true);
row1.put("考试日期", DateUtil.date());

Map<String, Object> row2 = new LinkedHashMap<>();
row2.put("姓名", "李四");
row2.put("年龄", 33);
row2.put("成绩", 59.50);
```

```
row2.put("是否合格", false);
row2.put("考试日期", DateUtil.date());

ArrayList<Map<String, Object>> rows = CollUtil.newArrayList(row1, row2);
```

写出数据：

```
// 通过工具类创建writer
ExcelWriter writer = ExcelUtil.getWriter("d:/writeMapTest.xlsx");
// 合并单元格后的标题行，使用默认标题样式
writer.merge(row1.size() - 1, "一班成绩单");
// 一次性写出内容，使用默认样式
writer.write(rows);
// 关闭writer，释放内存
writer.close();
```

效果：

图片地址：https://static.oschina.net/uploads/img/201711/12134150_BDDT.png

3. 写出Bean数据

定义Bean:

```
public class TestBean {
private String name;
private int age;
private double score;
private boolean isPass;
private Date examDate;

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public int getAge() {
return age;
}

public void setAge(int age) {
```

```
this.age = age;
}

public double getScore() {
return score;
}

public void setScore(double score) {
this.score = score;
}

public boolean isPass() {
return isPass;
}

public void setPass(boolean isPass) {
this.isPass = isPass;
}

public Date getExamDate() {
return examDate;
}

public void setExamDate(Date examDate) {
this.examDate = examDate;
}
}
````

构造数据：

```java
TestBean bean1 = new TestBean();
bean1.setName("张三");
bean1.setAge(22);
bean1.setPass(true);
bean1.setScore(66.30);
bean1.setExamDate(DateUtil.date());

TestBean bean2 = new TestBean();
bean2.setName("李四");
bean2.setAge(28);
```

```
bean2.setPass(false);
bean2.setScore(38.50);
bean2.setExamDate(DateUtil.date());

List<TestBean> rows = CollUtil.newArrayList(bean1, bean2);
```

写出数据：

```
// 通过工具类创建writer
ExcelWriter writer = ExcelUtil.getWriter("d:/writeBeanTest.xlsx");
// 合并单元格后的标题行，使用默认标题样式
writer.merge(4, "一班成绩单");
// 一次性写出内容，使用默认样式
writer.write(rows);
// 关闭writer，释放内存
writer.close();
```

效果：

图片地址：https://static.oschina.net/uploads/img/201711/12143029_3B2E.png

4. 自定义Bean的key别名（排序标题）

在写出Bean的时候，我们可以调用ExcelWriter对象的addHeaderAlias方法自定义Bean中key的别名，这样就可以写出自定义标题了（例如中文）。

写出数据：

```
// 通过工具类创建writer
ExcelWriter writer = ExcelUtil.getWriter("d:/writeBeanTest.xlsx");

//自定义标题别名
writer.addHeaderAlias("name", "姓名");
writer.addHeaderAlias("age", "年龄");
writer.addHeaderAlias("score", "分数");
writer.addHeaderAlias("isPass", "是否通过");
writer.addHeaderAlias("examDate", "考试时间");

// 合并单元格后的标题行，使用默认标题样式
writer.merge(4, "一班成绩单");
// 一次性写出内容，使用默认样式
writer.write(rows);
// 关闭writer，释放内存
writer.close();
```

效果：

图片地址：https://static.oschina.net/uploads/img/201808/01220010_Ybbw.png

> 提示（since 4.1.5）
>
> 默认情况下Excel中写出Bean字段不能保证顺序，此时可以使用addHeaderAlias方法设置标题别名，Bean的写出顺序就会按照标题别名的加入顺序排序。
>
> 如果不需要设置标题但是想要排序字段，请调用writer.addHeaderAlias("age", "age")设置一个相同的别名就可以不更换标题。
>
> 未设置标题别名的字段不参与排序，会默认排在前面。

5. 写出到流

```
// 通过工具类创建writer，默认创建xls格式
ExcelWriter writer = ExcelUtil.getWriter();
//创建xlsx格式的
//ExcelWriter writer = ExcelUtil.getWriter(true);
// 一次性写出内容，使用默认样式
writer.write(rows);
//out为OutputStream，需要写出到的目标流
writer.flush(out);
// 关闭writer，释放内存
writer.close();
```

6. 写出到客户端下载（写出到Servlet）

```
// 通过工具类创建writer，默认创建xls格式
ExcelWriter writer = ExcelUtil.getWriter();
// 一次性写出内容，使用默认样式
writer.write(rows);
//out为OutputStream，需要写出到的目标流

//response为HttpServletResponse对象
response.setContentType("application/vnd.ms-excel;charset=utf-8");
//test.xls是弹出下载对话框的文件名，不能为中文，中文请自行编码
response.setHeader("Content-Disposition","attachment;filename=test.xls");
ServletOutputStream out=response.getOutputStream();

writer.flush(out);
// 关闭writer，释放内存
writer.close();
```

> 注意

ExcelUtil.getWriter()默认创建xls格式的Excel，因此写出到客户端也需要自定义文件名为XXX.xls，否则会出现文件损坏的提示。

若想生成xlsx格式，请使用ExcelUtil.getWriter(true)创建。

自定义Excel

1. 设置单元格背景色

```
ExcelWriter writer = ...;

// 定义单元格背景色
StyleSet style = writer.getStyleSet();
// 第二个参数表示是否也设置头部单元格背景
style.setBackgroundColor(IndexedColors.RED, false);
```

2. 自定义字体

```
ExcelWriter writer = ...;
//设置内容字体
Font font = writer.createFont();
font.setBold(true);
font.setColor(Font.COLOR_RED);
font.setItalic(true);
//第二个参数表示是否忽略头部样式
writer.getStyleSet().setFont(font, true);
```

3. 写出多个sheet

```
//初始化时定义表名
ExcelWriter writer = new ExcelWriter("d:/aaa.xls", "表1");
//切换sheet，此时从第0行开始写
writer.setSheet("表2");
...
writer.setSheet("表3");
...
```

4. 更详细的定义样式

在Excel中，由于样式对象个数有限制，因此Hutool根据样式种类分为4个样式对象，使相同类型的单元格可以共享样式对象。样式按照类别存在于StyleSet中，其中包括：

- 头部样式 headCellStyle
- 普通单元格样式 cellStyle
- 数字单元格样式 cellStyleForNumber
- 日期单元格样式 cellStyleForDate

其中cellStyleForNumber cellStyleForDate用于控制数字和日期的显示方式。

因此我们可以使用以下方式获取CellStyle对象自定义指定种类的样式：

```
StyleSet style = writer.getStyleSet();
CellStyle cellStyle = style.getHeadCellStyle();
...
```

Excel大数据生成-BigExcelWriter

介绍

对于大量数据输出，采用ExcelWriter容易引起内存溢出，因此有了BigExcelWriter，使用方法与ExcelWriter完全一致。

使用

```
List<?> row1 = CollUtil.newArrayList("aa", "bb", "cc", "dd", DateUtil.date(),
3.22676575765);
List<?> row2 = CollUtil.newArrayList("aa1", "bb1", "cc1", "dd1", DateUtil.date(),
250.7676);
List<?> row3 = CollUtil.newArrayList("aa2", "bb2", "cc2", "dd2", DateUtil.date(), 0.111);
List<?> row4 = CollUtil.newArrayList("aa3", "bb3", "cc3", "dd3", DateUtil.date(), 35);
List<?> row5 = CollUtil.newArrayList("aa4", "bb4", "cc4", "dd4", DateUtil.date(), 28.00);

List<List<?>> rows = CollUtil.newArrayList(row1, row2, row3, row4, row5);

BigExcelWriter writer= ExcelUtil.getBigWriter("e:/xxx.xlsx");
// 一次性写出内容，使用默认样式
writer.write(rows);
// 关闭writer，释放内存
writer.close();
```

系统调用(Hutool-system)

系统属性调用-SystemUtil

概述

此工具是针对System.getProperty(name)的封装，通过此工具，可以获取如下信息：

Java Virtual Machine Specification信息

```
SystemUtil.getJvmSpecInfo();
```

Java Virtual Machine Implementation信息

```
SystemUtil.getJvmInfo();
```

Java Specification信息

```
SystemUtil.getJavaSpecInfo();
```

Java Implementation信息

```
SystemUtil.getJavaInfo();
```

Java运行时信息

```
SystemUtil.getJavaRuntimeInfo();
```

系统信息

```
SystemUtil.getOsInfo();
```

用户信息

```
SystemUtil.getUserInfo();
```

当前主机网络地址信息

```
SystemUtil.getHostInfo();
```

运行时信息，包括内存总大小、已用大小、可用大小等

```
SystemUtil.getRuntimeInfo();
```

图形验证码(Hutool-captcha)

概述

由来

由于对验证码需求量巨大，且我之前项目中有所积累，因此在Hutool中加入验证码生成和校验功能。

介绍

验证码功能位于cn.hutool.captcha包中，核心接口为ICaptcha，此接口定义了以下方法：

- createCode 创建验证码，实现类需同时生成随机验证码字符串和验证码图片
- getCode 获取验证码的文字内容
- verify 验证验证码是否正确，建议忽略大小写
- write 将验证码写出到目标流中

其中write方法只有一个OutputStream，ICaptcha实现类可以根据这个方法封装写出到文件等方法。

AbstractCaptcha为一个ICaptcha抽象实现类，此类实现了验证码文本生成、非大小写敏感的验证、写出到流和文件等方法，通过继承此抽象类只需实现createImage方法定义图形生成规则即可。

实现类

LineCaptcha 线段干扰的验证码

生成效果大致如下：

图片地址：https://static.oschina.net/uploads/img/201712/16113708_B8Hu.png

贴栗子：

```
//定义图形验证码的长和宽
LineCaptcha lineCaptcha = CaptchaUtil.createLineCaptcha(200, 100);

//图形验证码写出，可以写出到文件，也可以写出到流
lineCaptcha.write("d:/line.png");
//输出code
Console.log(lineCaptcha.getCode());
//验证图形验证码的有效性，返回boolean值
```

```
lineCaptcha.verify("1234");

//重新生成验证码
lineCaptcha.createCode();
lineCaptcha.write("d:/line.png");
//新的验证码
Console.log(lineCaptcha.getCode());
//验证图形验证码的有效性，返回boolean值
lineCaptcha.verify("1234");
```

CircleCaptcha 圆圈干扰验证码

图片地址：https://static.oschina.net/uploads/img/201712/16113738_eqt9.png

贴栗子：

```
//定义图形验证码的长、宽、验证码字符数、干扰元素个数
CircleCaptcha captcha = CaptchaUtil.createCircleCaptcha(200, 100, 4, 20);
//CircleCaptcha captcha = new CircleCaptcha(200, 100, 4, 20);
//图形验证码写出，可以写出到文件，也可以写出到流
captcha.write("d:/circle.png");
//验证图形验证码的有效性，返回boolean值
captcha.verify("1234");
```

ShearCaptcha 扭曲干扰验证码

图片地址：https://static.oschina.net/uploads/img/201712/16113807_sICp.png

贴栗子：

```
//定义图形验证码的长、宽、验证码字符数、干扰线宽度
ShearCaptcha captcha = CaptchaUtil.createShearCaptcha(200, 100, 4, 4);
//ShearCaptcha captcha = new ShearCaptcha(200, 100, 4, 4);
//图形验证码写出，可以写出到文件，也可以写出到流
captcha.write("d:/shear.png");
//验证图形验证码的有效性，返回boolean值
captcha.verify("1234");
```

写出到浏览器（Servlet输出）

```
ICaptcha captcha = ...;
captcha.write(response.getOutputStream());
```

```
//Servlet的OutputStream记得自行关闭哦！
```