

# li's framework

by li

416133823 [limw@w.cn](mailto:limw@w.cn)

2012-09-11

# 目录

1. 前言.....	- 3 -
1.1 这是什么.....	- 3 -
2. 版本信息.....	- 3 -
2.1 当前版本.....	- 3 -
3. Hello World.....	- 3 -
3.1 Hello Mvc.....	- 3 -
3.2 Hello Ioc.....	- 4 -
3.3 Hello Dao.....	- 4 -
4. Ioc.....	- 5 -
4.1 Ioc 概述.....	- 5 -
4.2 用注解配置 Ioc.....	- 5 -
4.3 用 Xml 配置 Ioc.....	- 6 -
4.4 使用 Properties.....	- 6 -
4.5 Ioc 工具类.....	- 7 -
5. Dao.....	- 7 -
5.1 DataSource.....	- 7 -
5.2 AbstractDao.....	- 8 -
5.3 模板事务.....	- 9 -
5.4 Dao 注解.....	- 10 -
5.5 ActiveRecord.....	- 10 -
6. Mvc.....	- 11 -
6.1 URL 映射.....	- 11 -
6.2 AbstractAction.....	- 12 -
6.3 Context 工具类.....	- 12 -
6.4 参数适配.....	- 13 -
6.5 视图.....	- 14 -
6.6 国际化.....	- 14 -
7. Aop.....	- 15 -
7.1 Aop 基础.....	- 15 -
7.2 AopFilter.....	- 16 -
7.3 AopChian.....	- 16 -
7.4 @Aop.....	- 16 -
7.5 @Trans.....	- 17 -
8. Util 工具类.....	- 17 -
8.1 Log.....	- 17 -
8.2 Convert.....	- 17 -
8.3 Verify.....	- 18 -
8.4 Files.....	- 18 -
8.5 Reflect.....	- 18 -
9. 其他.....	- 19 -

# 1. 前言

## 1.1 这是什么

li's framework，或者叫 li，是一个超轻量级的 J2EE 开发框架，包含 Ioc，Mvc，Dao 等的一整套简单便捷的实现。

# 2. 版本信息

## 2.1 当前版本

0.8 版，已经完成了绝大部分的功能，希望能够尽快进入 1.0。

# 3. Hello World

## 3.1 Hello Mvc

我们来尝试构建第一个使用 li's framework 开发的程序。

1. 打开 Eclipse，新建一个 Dynamic Web project，命名为 demo。
2. 把框架 JAR 包放入 lib 目录。
3. 修改 web.xml，添加 actionFilter：

```
<filter>
    <filter-name>actionFilter</filter-name>
    <filter-class>li.mvc.ActionFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>actionFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

4. 新建一个类，如下

```
@Bean
public class HelloWorld extends AbstractAction {
    @At("hello")
    public void hello() {
        super.write("hello,world!!!");
    }
}
```

```
    }
}
```

5. 运行这个项目，在浏览器地址栏中输入 `http://localhost:8080/demo/hello`，你会看到页面显示 `hello,world!!!`。

## 3.2 Hello Ioc

新建一个类，代码如下：

```
@Bean
public class HelloIoc {
    @Inject("123")
    public Integer num1;
    @Inject("456")
    public Integer num2;
}
```

测试

```
public static void main(String[] args) {
    HelloIoc helloIoc = Ioc.get(HelloIoc.class);
    System.out.println("sum = " + helloIoc.num1 + helloIoc.num2);
}
```

运行这个类，打印结果为：

```
sum = 579
```

## 3.3 Hello Dao

新建你的数据对象类

```
@Table
public class User {
    @Field
    private String username;
    @Field
    private String password;

    //Getters and Setters
}
```

新建你的数据操作类

```
@Bean
public class UserDao extends AbstractDao<User>{ }
```

测试

```
public static void main(String[] args) {  
    UserDao userDao = Ioc.get(UserDao.class);  
    User user = new User();  
    user.setUsername("li");  
    user.setPassword("123");  
    userDao.save(user);  
}
```

就像这样，你的 user 已经被存入数据库了。框架会自动生成并执行这样的 SQL 语句：

```
INSERT INTO User (username,password) VALUES ('li','123')
```

**注意：**事实上，你需要先新建数据表并配置数据源，具体请参照 5.1。

## 4. Ioc

### 4.1 ioc 概述

到目前为止，你可以使用 Xml 和注解两种方式来配置 Ioc。

两种方式均支持注入引用对象以及注入基本数据类型的值。

### 4.2 用注解配置 ioc

使用注解配置 Ioc 的示例如下：

```
@Bean  
public class ForumAction {  
    @Inject  
    IForumService forumService;  
}
```

使用 @Bean 注解标记的类为一个 Bean，你也可以为这个 Bean 确定一个名字，像这样

@Bean("beanName")。

使用 @Inject 标记的字段为一个 Ioc 属性。按照上面例子中的配置，forumService 属性会被设置为框架中 IForumService 类型或其子类型的 Bean。@Inject 注解也可以有值：@Inject("beanName")。此时，当字段为基本类型的时候，括号里的值将会被设值给这个属性（自动完成类型转换）。当字段类型不是基本类型时，这个字段将会被设置为一个 Bean，这个 Bean 名为 beanName，类型为属性类型或其子类型。

## 4.3 用 Xml 配置 Ioc

我们可以使用 Xml 来配置 Ioc，把你的 Xml 配置文件命名为以 config.xml 结尾的形式（如：config.xml，dao\_config.xml），并放在 SourceFolder 下，框架就会去找到它。

一个典型的 Xml Ioc 配置文件会像这样：

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <beans>
    <bean name="simple" class="li.dao.SimpleDataSource">
      <property name="url"
        value="jdbc:mysql://ip/db?characterEncoding=UTF-8"/>
      <property name="username" value="root"/>
      <property name="password"/>
    </bean>
    <bean class="demo.AccountDaoImpl">
      <property name="dataSource" value="simple"/>
    </bean>
  </beans>
</config>
```

所有的 Ioc 配置都在 config -> beans 下，每一个<bean></bean>标签标识一个 Bean。name 属性标识其名称，便于通过名称来查找到这个 Bean。当然，这并不是必须的，如果一个 Bean 没有被定义 name 的话，我们就只能通过 class 来查找。class 属性标识这个 bean 的类型。

每一个 bean 标签都可以有若干个 property 子标签，每一个 property 表示其所属 bean 的一个属性，name 为属性名称，value 为属性的值。

当一个属性的类型为基本数据类型时，这个属性会被设定为 value。否则，这个属性的值会是从 Ioc 上下文中查找到的符合这个属性类型的（当 property -> value 不为空时，名为 value 的）Bean。

## 4.4 使用 Properties

为了更好的组织配置信息，你可以使用 Properties 文件以键值对的形式配置相关信息，然后在 Xml 中引用其中的值。

注意：

你的 Properties 命名应当以 config.properties 结尾。

外部值引用只适用于 Xml 配置文件中 property 节点的 value 属性。

示例，config.xml 中：

```
<bean name="simple" class="li.dao.SimpleDataSource">
    <property name="url" value="${db.url}" />
    <property name="username" value="${db.username}" />
    <property name="password" value="${db.password}" />
</bean>
```

config.properties 中：

```
#DB
db.url=jdbc:mysql://127.0.0.1/db?characterEncoding=UTF-8
db.username=root
db.password=pwd
```

## 4.5 loc 工具类

li.ioc.Ioc 类提供了一些静态方法，用于从 Ioc 上下文中查找特定的 Bean：

```
public static <T> T get(String name)
```

从 Ioc 上下文中得到一个名为 name 的 Bean 的实例。

```
public static <T> T get(Class<T> type)
```

得到一个 Bean 的实例，这个 Bean 的类型为 type 或 type 的子类型。

```
public static <T> T get(Class<T> type, String name)
```

得到 type 和 name 同时符合的 Bean 的实例；如果 name 为空，则返回一个类型符合的。

# 5. Dao

## 5.1 DataSource

DataSource 其实也是一个普通的 Bean，你可以自由的选用 DBCP，C3P0 等数据源，通过 Xml Ioc 配置文件配置。同时，我们为您准备了 SimpleDataSource：一个 DataSource 的简单实现，供您测试时使用。

示例：

```
<bean name="c3p0"
    class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="jdbcUrl"
        value="jdbc:mysql://ip/db?characterEncoding=UTF-8"/>
    <property name="user" value="root"/>
    <property name="password" value=""/>
</bean>
```

```

        <property name="acquireIncrement" value="3"/>
        <property name="acquireRetryAttempts" value="30"/>
        <property name="acquireRetryDelay" value="1000"/>
    </bean>
    <bean name="simple" class="li.dao.SimpleDataSource">
        <property name="url"
            value="jdbc:mysql://ip/db?characterEncoding=UTF-8"/>
        <property name="username" value="root"/>
        <property name="password"/>
    </bean>

```

## 5.2 AbstractDao

这是一个泛型类。通常的，你的数据访问的类都应该继承于它。例如：

```
@Bean
```

```
public class UserDao extends AbstractDao<User>{    }
```

就这样，你的 UserDao 就已经具备了强大的数据操作能力。而且你不再需要进行繁琐的 SQL 组装，对象组装，异常处理等工作。

你只需要用如下的简单代码，就可以把一个数据对象 user 写入数据库：

```
userDao.save(user);
```

插入完成后，**user** 的 **ID** 属性将会被设置为刚刚插入数据时候自增 **ID** 的值。

你还可以在 UserDao 中新增方法，使用简单的代码进行扩展。就像这样：

```
public User findByUsername(String username){
    return super.find("WHERE username=?", username);
}
```

**注意：**这里略去了建立 POJO 和数据源两个步骤，请参照 5.1 和 3.3 两节。

AbstractDao 方法中提供了一系列有可变个数的 Object 类型参数的方法，这个 Object 数组即是 SQL 查询的参数。

当参数不为 Map 类型时，传入参数将会第次替换 SQL 中的 '?' 占位符。

当参数为 Map 类型时，SQL 中的等于 Map 中某个 key 的字符序列将会被替换为 Map 中对应的值。

示例：

```

Map<?, ?> map = Convert.toMap(":uname", "li", ":eml", "li@w.cn");
dao.list(page, "WHERE username=:uname OR email=:eml", map);
// SELECT * FROM t_account WHERE username='li' OR email='li@w.cn'

dao.find("WHERE username=? OR email=?", "li", "li@w.cn");

```



```
// SELECT * FROM t_account WHERE username='li' OR email='li@w.cn'

dao.delete("WHERE username=:uname OR email=:eml OR id=?", map, 1);
// DELETE FROM t_account WHERE username='li' OR email='li@w.cn'
OR id='1'
```

## 5.3 模板事务

示例：

```
public static void main(String[] args) {
    AccountDaoImpl accountDao = Ioc.get(AccountDaoImpl.class);
    MemberDaoImpl memberDao = Ioc.get(MemberDaoImpl.class);
    new Trans() {
        public void run() {
            accountDao.update("set password='xyz' where id=5");
            memberDao.update("set id='abc' where true");
        }
    };
}
```

使用 Trans 包裹的数据操作方法会处在一个事务中。如果其中的一个操作失败（即 Statement execute 出现异常，会被框架捕获），其中所有的数据操作都将会被撤销，即使他们使用不同的 DataSource，不同的数据库。

形如 **new** Trans() 或者 **new** Trans(**true**) 的事务会自动执行，即在 Trans 完成初始化后即刻执行。以 **new** Trans(**false**) 形式创建的事务不会自动执行，你可以在创建这个事务后调用 go() 方法以执行之。

当使用手动提交的事务模板时，你可以使用 Trans.set() 方法和 Trans.map() 方法进行事务模板内外传值。

```
User user = (User) new Trans(false) {
    public void run() {
        userDao.update("SET email=:email WHERE username=:username",
            map());
        map().put("user",
            userDao.find("WHERE username=?",
                map().get(":username")));
    }
}.set(
    Convert.toMap(":email", "tom@w.cn", ":username", "xiaoming")
).go().map().get("user");
```

Trans有一个success()方法, 返回一个Boolean 值。表示事务是否执行成功,true表示成功,false表示失败。

```
Boolean flag = new Trans() {  
    public void run() {  
        userDao.update("SET email='li@w.cn' WHERE 1=2");  
    }  
}.success();
```

当出现事务嵌套的情形时, 只有最外层的事务是起作用的, 被包裹的事务会“融化”在上层事务里。

## 5.4 Dao 注解

@Table 标记一个对象对应于某一张数据表。可以用 @Table("table\_name") 的形式定义表名。如未特别指定, 表名将会是对象类型的 getClass().getSimpleName()。

@Field 标记一个对应于数据表中一列的字段。某个属性上一个形如 @Field(id = true, value="id\_column") 的表示当前属性是一个 id, 数据表中对应的字段为 id\_column。id 默认为 false, value (列名) 默认为属性字段名。

## 5.5 ActiveRecord

除了以上面的方式借助 AbstractDao 操作数据库以外, 我们还提供了另外一种, 在某些情况下, 更为便捷的方式, 那就是 ActiveRecord:

```
@Bean  
@Table("t_user")  
public class User extends Record<User> {}
```

通过继承 Record, 你的 User 类已经同时拥有数据对象和 Dao 的功能。你甚至不需要写任何的 Field。

示例:

```
public static void main(String[] args) {  
    User userDao = Ioc.get(User.class);  
    for (User user : userDao.list(new Page())) {  
        System.out.println(user.get("username"));  
    }  
}
```

再看一个示例:

```

public static void main(String[] args) {
    User userDao = Ioc.get(User.class);
    String sql = "select * from user,msg where msg.uid=user.id";
    List<Record> records = userDao.list(new Page(), sql);
    for (Record record : records) {
        System.out.print(record.get("user_name"));
        System.out.print(record.get("msg_title"));
    }
}

```

是否你已经发现，使用 ActiveRecord 方式还可以进行多表操作。是的，你只需要提供正确的 sql 语句，查询的结果集将会被原样的转换为一个 Record 的 List。

你同样可以使用事务模板 Trans 来包裹 Record 的子类的数据操作方法，来进行事务控制。

Record 是 AbstractDao 的子类，拥有 AbstractDao 的全部特性。

## 6. Mvc

### 6.1 URL 映射

Action 类必须是一个使用 Xml 或者注解方式配置的 Bean。

Action 方法可以返回一个字符串以指定视图，或者返回 void。

使用 @At 注解来标记一个 Action 方法。就像 Hello Mvc 里面的一样。

@At 注解有两个属性，value 表示 action 请求路径，不指定则为 Action 方法名；method 表示请求类型，不指定则为任意类型。

例如：

```
@At(value = "login.do", method = "POST")
```

则表示其所在的 Action 方法对应于路径为 login.do 类型为 POST 的请求。

你可以使用正则表达式配置 Action 访问路径。

示例，一个 Action 方法如下：

```

@At("thread-([0-9]*)-([a-z]*).htm")
public void test() {
    for (String param : Context.pathParams()) {
        Context.write(param);
    }
}

```

当请求地址为 thread-123-abc.htm 时，这个 Action 方法将会执行，页面将会输出

123abc。

你可以使用任意的正则表达式（当然，得是正确的）来配置的 `Action` 地址。如果你想要从路径中取得一些参数的话，需要将路径中可变部分（即路径参数部分）用小括号包裹，就像上面的例子。

## 6.2 AbstractAction

你可以选择让你的 `Action` 类继承 `li.mvc.AbstractAction`，便可以方便的在你的 `Action` 中使用 `AbstractAction` 提供的一系列封装方法。你可以利用他们得到 `request` 和 `response`，或者直接使用它的封装方法获取数据和处理视图。例如：

```
public HttpServletRequest getRequest()
```

得到当前请求的 `request`。

```
public String getParameter(String key)
```

从 `QueryString` 中获取指定 `key` 对应的值。

```
public <T> T get(Class<T> type, String preFix)
```

从 `request` 中获取对应的值生成一个 `type` 类型的对象。

```
public AbstractAction write(String content)
```

把 `content` 表示的内容，直接输出到页面上。

```
public void view(String path)
```

视图分发方法，允许你通过使用前缀的方式调用它，以处理不同的视图。例如：

```
view("forward:add.jsp")           forward 到 add.jsp
```

```
view("redirect:index.jsp")         redirect 到 add.jsp
```

```
view("freemarker:edit.htm")        使用 freemarker 渲染视图，模版路径为 edit.htm
```

```
view("velocity:list.htm")          使用 velocity 渲染视图，模版路径为 list.htm
```

```
view("write:你好")                 用 response 将 你好 写到页面
```

## 6.3 Context 工具类

`Context` 使用 `ThreadLocal` 包装了 `request` 和 `response`，且提供了 `AbstractAction` 中所有方法的静态封装方法。

利用它，你的 `Action` 类可以不继承 `AbstractAction`，你可以通过在 `Action` 方法中调用 `Context` 的静态方法，来获取前端数据和返回视图。

## 6.4 参数适配

你可以通过上面提到的利用 Context 和继承 AbstractAction 的方式,得到 request response 等对象, 或者利用他们的内置系列方法从 request 中直接获取数据。

你也可以为你的 Action 方法配置 request 和 response 两个参数, 你将可以以类似于 Servlet 的方式来处理参数和视图。

你还可以直接将你所需要从前端得到的数据作为 Action 方法的参数, 框架将会传入 request 中相应的值。

目前支持以下参数类型:

1. 基本数据类型如 String, Integer, Boolean, Date, 参见 `li.util.Verify.basicType(Class<?>)`, **请避免使用原始数据类型如 int, boolean;**
2. 基本数据类型的数组;
3. POJO 数据对象;
4. request response page。

你可以使用 **@Arg 注解** 标记一个 Action 方法参数, 以使框架通过适当的方式来适配这些参数。

@Arg 可以使用在 Action 方法的基本类型参数及其数组 和 POJO 等参数上。

如 `@Arg("user_count") Integer userCount`, @Arg 注解在基本类型参数或其数组上时, 表示参数取值于 QueryString 中 key 为 user\_count 的值, 框架会自动为你完成数据类型转换。没有注解时相当于 `@Arg(参数名)`。

如 `@Arg("user.") User user`, @Arg 注解在 POJO 数据对象类型的参数上时, 表示数据对象的每一个属性的值取于 QueryString 中 key 为 "user." + 属性名 的值。如果数据对象参数上没有注解 @Arg, 则相当于 `@Arg(参数名+".")`。

`@Arg("pn") Page page`, @Arg 注解使用在 Action 方法的 `li.dao.Page` 类型的参数上时。参数将被设值为 `getPage(String pn)` 的值。

举个例子:

```
@At("demo1")
public void demo1(String str1, /* 注入key为str1的参数 */
    @Arg("int1") Integer int1, /* 注入key为int1的参数 */
    Account account, /* 注入属性前缀为account.的Account */
    @Arg("pn") Page page, /* 注入key为pn的Page */
```

```

        HttpServletResponse response /* 注入response */)
    {
        super.getParameter(""); //用超类AbstractAction中方法得到参数
        Context.get(Account.class, "account."); // Context中静态方法
    }

```

## 6.5 视图

你可以使用上面提到的 `getRequest()`和 `getResponse()`方式得到 `request` 和 `response` 然后通过类似 `Servlet` 的方式处理视图的问题。

你也可以使用 `AbstractAction` , `Context` 中提供的 `view forward redirect freemaker velocity write` 系列方法。

你还可以使你的 `Action` 方法返回一个表示视图类型和路径的字符串。这也就相当于使用 `Context` 或 `AbstractAction` 中的 `view(String path)` 方法。返回字符串格式为 类型: 内容。如 `return "redirect:index.jsp";` 。

示例:

```

@At("demo2")
public String demo2() {
    return "demo2.jsp"; //通过方法返回值确定视图, 无前缀默认JSP视图
    //return "freemarker:demo2.htm"; //通过返回值的前缀指定视图类型
    //return forward("demo2.jsp"); //调用方法返回特定类型的视图
    //super.view("velocity:demo2.htm"); //AbstractAction视图方法
    //super.freemarker("demo2.htm"); //Action方法返回值为可以void
    //return Context.velocity("demo2.htm"); //Context中的静态方法
    //Context.write("使用Context中的静态方法将文本写到页面");
}

```

## 6.6 国际化

是这样的, 把你的国际化文件, 即 `properties` 文件放到 `Source Folder`, 然后就可以在页面上通过 `${lang["key"]}` 或 `${lang.key}` 访问到了。

你得注意:

1. 国际化文件的命名规则: 以 `.properties` 结尾, 前面由两部分组成, 第一部分为 `lang_` 第二部分为 Java 国际化代码, 例如 `en_US` , `zh_CN` 。你可以从 `java.util.Locale` 中得到他们, 例如: `lang_zh_CN.properties` 。

2. 国际化文件的内容为等号连接的键值对, 就像这样:

#国际化文件，等号前面是key，后面是value，本行是注释  
name=姓名  
sex=性别  
welcome=欢迎

## 7. Aop

### 7.1 Aop 基础

框架的 Aop 功能依赖于 cglib。你需要在你的项目中加入 cglib-nodep-2.2.3.jar，才可以使用 Aop 功能，否则你的所有 Aop 将会是无效的。当然，如果你不需要使用 Aop 功能，可以不必管它。

开发 Aop 中，你需要接触到 AopFilter，AopChain 和 @Aop。

示例：

```
@Bean
public class User {
    @Trans
    @Aop(LogFilter.class)
    public String sayHi(String msg1, String msg2) {
        System.out.println("user say hi");
        return msg1 + "\t" + msg2;
    }
}

@Bean
public class LogFilter implements AopFilter {
    @Inject("hello you")
    private String msg;

    public void doFilter(AopChain chain) {
        System.out.println("log before " + msg); //
        System.out.println(chain.getTarget()); // 目标对象, 已被Aop
        System.out.println(chain.getMethod()); // 目标方法
        System.out.println(chain.getArgs()); // 方法参数列表
        chain.doFilter(); // 执行目标方法或者下一个AopFilter
        System.out.println(chain.getResult()); // 方法结果
        System.out.println("log after");
    }
}
```

你完全可以以 Servlet 中的 Filter 来理解这里的 AopFilter，并且以类似的方式，

方便的实现 Aop 功能。

## 7.2 AopFilter

AopFilter 接口定义了一个 **public void** doFilter(AopChain chain) 方法，你可以通过设计这个接口的实现类来实现 Aop。通过在 doFilter 方法中加入相关代码实现 Aop 功能。

## 7.3 AopChian

AopChain 类似于 ServletFilter 中的 Chain，但这里提供了更多方法。

**public** AopChain doFilter()

执行 Aop 链中的下一个 AopFilter，或者执行目标方法。

**public** Object getTarget()

返回方法宿主对象，被 Aop 处理过的，是原始类型的子类。

**public** Method getMethod()

返回方法被 Aop 的对象。

**public** Object[] getArgs()

返回方法参数列表。

**public** AopChain setArgs(Object[] args)

设置方法参数，你可以使用此方法使目标方法以你于此处给定的参数执行，在 doFilter() 之前才有效，**慎用**。

**public** Object getResult()

获得方法返回值，在 doFilter() 后才有效。

**public** AopChain setResult(Object result)

修改方法的返回值，其后不再 doFilter() 才有效，**慎用**。

## 7.4 @Aop

定义好 AopFilter 的实现类后，可以通过在目标方法上添加 @Aop 注解来使用这些 AopFilter。

**注意：**你的 AopFilter 必须是一个通过注解或者 Xml 等方式配置的一个 IocBean。AopFilter 中可以使用 Ioc 注入，但 AopFilter 的实现类的方法不能 @Aop。

@Aop 注解有一个参数，为 AopFilter 的子类类型数组。框架将会使用这些类型，通过



按照类型匹配的方式，从 Ioc 环境中找到对应的 Bean。

## 7.5 @Trans

我们提供了一个基于 Aop 的事务注解，可以帮助你更加方便的进行事务管理。你可以不再使用事务模板，只需要在需要事务支持的方法上添加@Trans 注解即可。

注意：需要 Aop 支持，即需要 cglib。

# 8. Util 工具类

li.util 包中包含了一些工具类，在某些情形下，他们可以让你更方便的完成某些功能。

## 8.1 Log

我们提供了一个日志工具类，li.util.Log。他会自动适配 Log4j，当 Log4j 可以使用时，他调用 Log4j 输出日志。否则，他会把日志内容打印到控制台。

Log 的使用示例：

```
public class T {  
    public static final Log log = Log.init();  
    public static void main(String[] args) {  
        log.info("输出info级别的日志信息");  
        log.debug("debug级别日志");  
    }  
}
```

## 8.2 Convert

Convert 类提供了一些静态方法，方便你进行一些转换的操作。如：

```
public static Map<Object, Object> toMap(Object... items)
```

将数组转换为 Map，奇数位为 key，偶数位为 value； items 必须为偶数个。

```
public static String toMD5(String input)
```

将字符串使用 MessageDigest 进行一次 MD5 加密后返回。

```
public static String toJson(Object target)
```

将对象，对象的数组或者 Connection 转换为 Json。

```
public static <T> List<T> fromJson(Class<T> type, String json)
```

将 Json 字符串转换为 type 类型的对象的 List。

## 8.3 Verify

这是一个验证工具类，提供了一系列常用的用于验证的静态方法。如：

```
public static Boolean basicType(Class<?> type)
```

判断类型 type 是否是基本类型。

```
public static Boolean regex(String input, String regex)
```

判断 input 字符串是否符合 regex 正则表达式。

## 8.4 Files

文件操作的工具类。提供了一些文件操作的静态方法，例如：

```
public static List<String> list(File file, String regex, Boolean  
increase)
```

在 file 代表的目录中搜索文件名符合 regex 的文件，increase 表示是否递进搜索（搜索子文件夹）。

```
public static Object xpath(Object document, String xpath, QName  
returnType)
```

计算指定上下文 document 中的 XPath 表达式并返回指定类型 returnType 的结果。

```
public static String read(File file)
```

以字符流的方式读取一个文本文件并返回内容。

```
public static void write(File file, String content)
```

将 content 字符串写到 file 文件。

## 8.5 Reflect

Reflect 提供了一系列的经过包装过的反射方法。如：

```
public static <T> T born(Class<T> type, Object... args)
```

生成 type 对象的一个实例，args 为构造函数参数列表，可以没有。

```
public static Object invoke(Object target, String methodName,  
Object... args)
```

执行 target 对象的 methodName 方法，args 为参数列表，可以没有。

```
public static Object get(Object target, String fieldName)
```

返回 target 对象的 fieldName 属性的值。

```
public static void set(Object target, String fieldName, Object value)
```

设值 target 对象的 fieldName 属性的值为 value。

```
public static <T> T copy(Object src, T dest)
```

复制 src 对象的每一个属性的值给 dest 对象。

## 9. 其他

我们提供了几个基类或基接口，你可以选择使用：

**IBaseDao<T>**：基础 Dao 接口，定义了几个基础的方法，用户可以继承并扩展它形成自己的 Dao 接口。当然，也可以不。

**IBaseService<T>**：基础 Service 接口，定义了几个基础的方法，用户可以继承并扩展它形成自己的 Service 接口。当然，也可以不。

**AbstraceService<T>**：一个抽象的 Service 的实现类，实现了几个基础的方法，用户可以继承并扩展它形成自己的 Service 类。你可以重写他的 `getDao()` 方法，不然他会使用 `Ioc.get(Class<T> type, Type genericType)` 方法，从 `IocContext` 中搜索 `AbstractDao` 的子类中泛型类型为 T 的类。

我们设计了一些 **SQL** 扩展特性，示例：

```
//可以省略 SELECT * FROM table 部分
dao.list(page, "WHERE true");
dao.list(page, "SELECT * FROM table WHERE true");
dao.list(page, "SELECT * FROM table");

//可以省略 SELECT * FROM table 部分
dao.find("WHERE id=?", 1);
dao.find("SELECT * FROM table WHERE id=?", 1);

//可以省略 SELECT COUNT(*) FROM table 部分
dao.count("WHERE id=?", 1);
dao.count("SELECT * FROM table WHERE id=?", 1);
dao.count("SELECT COUNT(*) FROM table WHERE id=?", 1);

//可以省略 DELETE FROM table 部分
dao.delete("WHERE id=?", 1);
dao.delete("DELETE FROM table WHERE id=?", 1);
```

```
//可以省略 UPDATE table 部分
dao.update("SET name='NNN' WHERE id=?", 1);
dao.update("UPDATE table SET name='NNN' WHERE id=?", 1);

//你可以通过SELECT table.# AS t_#的形式为一个表的所有列设值别名

/*处理为 SELECT table.column1 AS t_column1,
    table.column2 AS t_column2
    FROM table LIMIT 0,20*/
dao.list(page, "SELECT table.# AS t_# FROM table");

/*处理为 SELECT t_thread.id,t_thread.subject,
    t_thread.content,t_thread.forum_id,
    t_thread.member_id,t_thread.status,
    t_member.id AS member_id,
    t_member.status AS member_status,
    t_member.name AS member_name,
    t_member.account_id AS member_account_id
    FROM t_thread,t_member
    WHERE t_thread.forum_id='1'
    AND t_thread.member_id=t_member.id
    LIMIT 0,20*/
dao.list(page, "SELECT t_thread.#, " +
    "t_member.# AS member_# " +
    "FROM t_thread,t_member " +
    "WHERE t_thread.forum_id=? " +
    "AND t_thread.member_id=t_member.id", 1);
```

更多信息请参照 API 文档。