

配置变量

目录

1. 概述	2
2. 基本原理	2
2.1. 基本分层	2
2.2. 专门环境	4
2.2.1. JNDI 变量	4
2.2.2. 环境变量	4
2.2.3. 配置文件	7
3. 具体情形	7
3.1. 纯 spring	7
3.1.1. 一般情形	7
3.1.1.1. environmentProperties	8
3.1.1.2. systemProperties	8
3.1.1.3. systemEnvironment	9
3.1.1.4. localProperties	9
3.1.2. web 容器内	11
3.1.2.1. servletConfigInitParams	11
3.1.2.2. servletContextInitParams	12
3.1.2.3. jndiProperties	13
3.2. spring boot 环境	14
3.2.1. spring boot 环境下相对纯 spring 环境的变更	14
3.2.1.1. yaml 文件支持	14
3.2.1.2. 可选的配置文件引入	15
3.2.1.3. 支持通配符的配置文件引入	15
3.2.1.4. 支持缺少扩展名的文件载入	15
3.2.1.5. 支持文件系统式的复合配置	15
3.2.1.6. 配置变量中可以引入配置变量	16
3.2.1.7. 基于单文件的多环境配置	16
3.2.1.8. 宽松绑定	16
3.2.1.9. 配置随机值	16
3.2.1.10. 配置变量与 bean 绑定	17
3.2.2. spring boot 的配置源	17
3.2.2.1. DevtoolsGlobalSetting	20
3.2.2.2. testPropertySource	21
3.2.2.3. properties attribute on @SpringBootTest	22
3.2.2.4. command line arguments	22
3.3. spring cloud 环境	22
4. 其他	22
5. 作业	22

1. 概述

本文介绍 **java** 基于 **spring** 框架构建的 **java** 程序的配置变量。这里的配置变量使用如下所示：

基于注解使用配置变量：

```
@Value("${sample_param}")  
private String sampleParam;
```

基于 **xml** 使用配置变量：

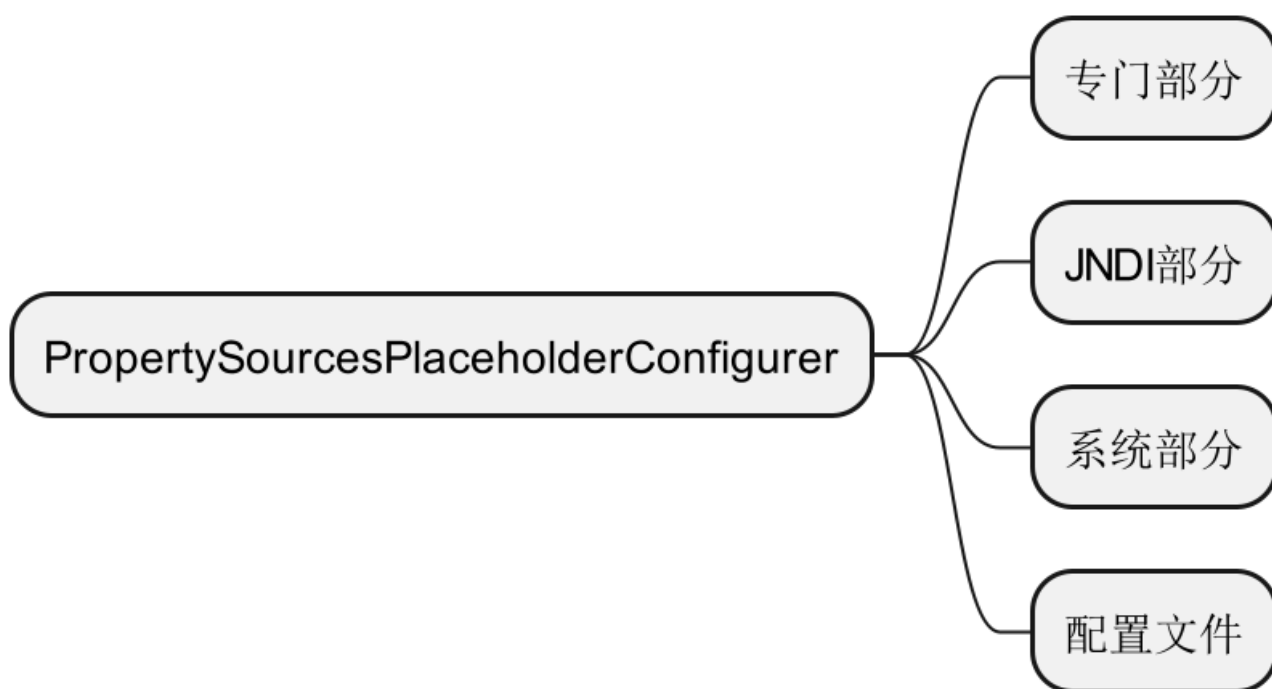
```
<bean class="com.yourCompany.yourApp.SampleBean">  
    <property name="sampleParam" value="${sample_param}"/>  
</bean>
```

简单的说，本文介绍 **`${sample_param}`** 的值由哪些因素决定，并介绍这些因素的使用规则以及如何确定到底是哪些因素决定了其最终值。

本文主要参考 **spring 5**，**spring boot 2** 以及对应的 **spring cloud** 版本的文献。

2. 基本原理

2.1. 基本分层



spring 的配置变量大致可以分为上述四套，其基本规则为上层覆盖下层（与常识不同，不是离应用越近越生

效，但本质是上层找到就不会到下层查找）。即若 环境变量 设置 `sample_param=a`, JNDI 环境 设置 `sample_param=b`, 配置文件 设置 `sample_param=c`, 则在 `spring` 中取到的 `${sample_param}` 值为 `b` 来自 JNDI 变量。

```
2022-10-24 23:17:58,331 TRC [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Searching for key 'jdbc_username' in
PropertySource 'environmentProperties'
2022-10-24 23:17:58,331 TRC [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Searching for key 'jdbc_username' in
PropertySource 'servletConfigInitParams'
2022-10-24 23:17:58,331 TRC [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Searching for key 'jdbc_username' in
PropertySource 'servletContextInitParams'
2022-10-24 23:17:58,331 TRC [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Searching for key 'jdbc_username' in
PropertySource 'jndiProperties'
...
2022-10-24 23:17:58,331 TRC [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Searching for key 'jdbc_username' in
PropertySource 'systemProperties'
2022-10-24 23:17:58,331 TRC [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Searching for key 'jdbc_username' in
PropertySource 'systemEnvironment'
2022-10-24 23:17:58,331 DBG [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Could not find key 'jdbc_username' in any
property source
2022-10-24 23:17:58,331 TRC [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Searching for key 'jdbc_username' in
PropertySource 'localProperties'
2022-10-24 23:17:58,331 DBG [localhost-startStop-1]
o.s.c.e.PropertySourcesPropertyResolver Found key 'jdbc_username' in PropertySource
'localProperties' with value of type String
2022-10-24 23:17:58,331 TRC [localhost-startStop-1] o.s.u.PropertyPlaceholderHelper
Resolved placeholder 'jdbc_username'
```

以上日志反应了 `spring` 解析名为 `jdbc_username` 变量时的过程，略去了 jndi 查询的详细部分，后续说明。涉及到的各配置源及说明如下。

表 1. 配置变量源

序号	名称	层次	说明
1	environmentProperties	专门环境	其配置入口需要写入 java 代码中，一般不作为配置源
2	servletConfigInitParams	专门环境	其配置入口在 web.xml 中，一般不作为配置源
3	servletContextInitParams	专门环境	其配置入口在 web.xml 中，一般不作为配置源
4	jndiProperties	JNDI环境	配置入口在容器，可以作为配置源

序号	名称	层次	说明
5	systemProperties	环境变量	java 程序特有的环境变量中间层，视为环境变量
6	systemEnvironment	环境变量	真正的环境变量。
7	localProperties	配置文件	应用中的配置文件



这里仅做简单说明，后续会有更详细的说明。

2.2. 专门环境

2.2.1. JNDI 变量

```
2022-10-24 23:55:46,456 DBG [localhost-startStop-1] o.s.j.JndiTemplate Looking up
JNDI object with name [java:comp/env/jdbc_username]
2022-10-24 23:55:46,457 DBG [localhost-startStop-1] o.s.j.JndiLocatorSupport
Converted JNDI name [java:comp/env/jdbc_username] not found - trying original name
[jdbc_username]. javax.naming.NameNotFoundException: Name [jdbc_username] is not bound
in this Context. Unable to find [jdbc_username].
2022-10-24 23:55:46,457 DBG [localhost-startStop-1] o.s.j.JndiTemplate Looking up
JNDI object with name [jdbc_username]
2022-10-24 23:55:46,457 DBG [localhost-startStop-1] o.s.j.JndiPropertySource JNDI
lookup for name [jdbc_username] threw NamingException with message: Name
[jdbc_username] is not bound in this Context. Unable to find [jdbc_username]..
Returning null.
```

如上，虽然 spring 的文档上称 spring 会在 `java:comp/env/` 下查找指定变量，但事实上其也会直接查找对应变量。

2.2.2. 环境变量

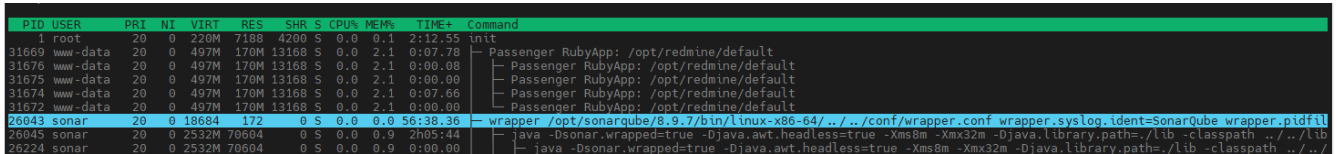
进程之间的关系

```
Failed to generate image: Could not find the 'mmdc' executable in PATH; add it to the
PATH or specify its location using the 'mmdc' document attribute
graph TB
A("init pid=1") --> B1("service1");
A --> B2("service2");
A --> C1("user1 shell");
A --> C2("user2 shell");
A --> C3("user3 shell");

C1 -->|fork| P1("java ...");
C1 -->|fork| P2("bash ...");
```

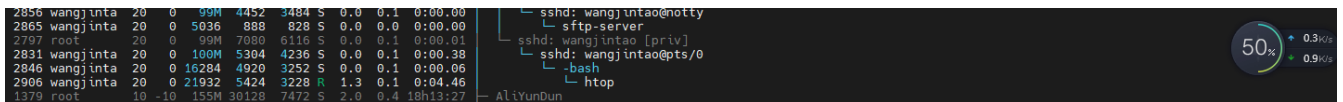
```
C1 -->|fork| P3("top ...");
```

谈环境变量必须谈进程，环境变量是从进程引出来的概念。以 linux 为例：运行时的 linux 多任务环境应视为一颗进程树，其根是 init 进程或 systemd 进程，进程号为 1，其直接子进程一般是各种系统服务。用户的 shell 进程也是其子进程（不一定是直接子进程）。如下图：



PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1	root	20	0	229M	7188	4200	S	0.0	0.1	2:12.55	init
31669	www-data	20	0	497M	170M	13168	S	0.0	2.1	0:07.78	Passenger RubyApp: /opt/redmine/default
31676	www-data	20	0	497M	170M	13168	S	0.0	2.1	0:00.08	Passenger RubyApp: /opt/redmine/default
31675	www-data	20	0	497M	170M	13168	S	0.0	2.1	0:00.00	Passenger RubyApp: /opt/redmine/default
31674	www-data	20	0	497M	170M	13168	S	0.0	2.1	0:07.66	Passenger RubyApp: /opt/redmine/default
31672	www-data	20	0	497M	170M	13168	S	0.0	2.1	0:00.00	Passenger RubyApp: /opt/redmine/default
26043	sonar	20	0	18684	172	0	S	0.0	0.0	56:38.36	wrapper /opt/sonarqube/8.9.7/bin/linux-x86-64/../../conf/wrapper.conf wrapper.syslog.ident=SonarQube wrapper.pidfile
26045	sonar	20	0	2532M	70604	0	S	0.0	0.9	2h05:44	java -Dsonar.wrapped=true -Djava.awt.headless=true -Xms8m -Xmx32m -Djava.library.path=/lib -classpath ../lib
26224	sonar	20	0	2532M	70604	0	S	0.0	0.9	0:00.00	java -Dsonar.wrapped=true -Djava.awt.headless=true -Xms8m -Xmx32m -Djava.library.path=/lib -classpath ../lib

图 1. init 与服务进程



PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2856	wangjintao	20	0	99M	4452	3484	S	0.0	0.1	0:00.00	sshd: wangjintao@notty
2865	wangjintao	20	0	5036	888	828	S	0.0	0.0	0:00.00	sftp-server
2797	root	20	0	99M	7080	6116	S	0.0	0.1	0:00.01	sshd: wangjintao [priv]
2831	wangjintao	20	0	100M	5304	4236	S	0.0	0.1	0:00.38	sshd: wangjintao@pts/0
2846	wangjintao	20	0	16284	4920	3252	S	0.0	0.1	0:00.06	-bash
2906	wangjintao	20	0	21932	5424	3228	R	1.3	0.1	0:04.46	httpd
1379	root	10	-10	155M	30128	7472	S	2.0	0.4	18h13:27	AllyunDun

图 2. 用户进程

在这个树中，父进程创建子进程时会隐式传参，这部分隐式参数就是环境变量。其原型如下：

```
int main(int argc,char* argv[],char* env[])
```

调用的函数见 linux 定义的 `execve` 系统调用。



环境变量在进程创建的过程中被复制了一份，所以我们应该认为父进程向子进程传递的是一份自身环境变量的副本。

在 linux 下，查看进程号为 `pid` 的环境变量值最稳妥方法为查看 `/proc/<pid>/environ` 的内容。建议使用如下命令：

```
od -A x -t x1cz /proc/<pid>/environ
```

输出如下：

```
000000 55 53 45 52 3d 77 61 6e 67 6a 69 6e 74 61 6f 00
          U  S  E  R  =  w  a  n  g  j  i  n  t  a  o  \0
>USER=wangjintao.<
000010 4c 4f 47 4e 41 4d 45 3d 77 61 6e 67 6a 69 6e 74
          L  O  G  N  A  M  E  =  w  a  n  g  j  i  n  t
>LOGNAME=wangjint<
000020 61 6f 00 48 4f 4d 45 3d 2f 68 6f 6d 65 2f 77 61
          a  o  \0  H  O  M  E  =  /  h  o  m  e  /  w  a
>ao.HOME=/home/wa<
000030 6e 67 6a 69 6e 74 61 6f 00 50 41 54 48 3d 2f 75
          n  g  j  i  n  t  a  o  \0  P  A  T  H  =  /  u
>ngjintao.PATH=/u<
000040 73 72 2f 6c 6f 63 61 6c 2f 73 62 69 6e 3a 2f 75
```

```

s r / l o c a l / s b i n : / u
>sr/local/sbin:/u<
000050 73 72 2f 6c 6f 63 61 6c 2f 62 69 6e 3a 2f 75 73
s r / l o c a l / b i n : / u s
>sr/local/bin:/us<
000060 72 2f 73 62 69 6e 3a 2f 75 73 72 2f 62 69 6e 3a
r / s b i n : / u s r / b i n :
>r/sbin:/usr/bin:<
000070 2f 73 62 69 6e 3a 2f 62 69 6e 3a 2f 75 73 72 2f
/ s b i n : / b i n : / u s r /
>/sbin:/bin:/usr/<
000080 67 61 6d 65 73 3a 2f 75 73 72 2f 6c 6f 63 61 6c
g a m e s : / u s r / l o c a l
>games:/usr/local<
000090 2f 67 61 6d 65 73 3a 2f 73 6e 61 70 2f 62 69 6e
/ g a m e s : / s n a p / b i n
>/games:/snap/bin<
0000a0 00 4d 41 49 4c 3d 2f 76 61 72 2f 6d 61 69 6c 2f
\0 M A I L = / v a r / m a i l /
>.MAIL=/var/mail/<
0000b0 77 61 6e 67 6a 69 6e 74 61 6f 00 53 48 45 4c 4c
w a n g j i n t a o \0 S H E L L
>wangjintao.SHELL<
0000c0 3d 2f 62 69 6e 2f 62 61 73 68 00 53 53 48 5f 43
= / b i n / b a s h \0 S S H _ C
>=/bin/bash.SSH_C<
0000d0 4c 49 45 4e 54 3d 31 32 33 2e 31 32 33 2e 31 30
L I E N T = 1 2 3 . 1 2 3
>LIENT=123.123.10<
0000e0 30 2e 31 33 31 20 33 33 34 39 20 32 32 00 53 53
0 . 1 3 1 3 3 4 9 2 2 \0 S S >0.131 3349
22.SS<
0000f0 48 5f 43 4f 4e 4e 45 43 54 49 4f 4e 3d 31 32 33
H _ C O N N E C T I O N = 1 2 3
>H_CONNECTION=123<
000100 2e 31 32 33 2e 31 30 30 2e 31 33 31 20 33 33 34
. 1 2 3 . 1 0 0 . 1 3 1 >.123.100.131
334<
000110 39 20 31 37 32 2e 33 31 2e 32 33 30 2e 37 37 20
9 1 7 2 . 3 1 . 2 3 0 . 7 7 >9
172.31.230.77 <
000120 32 32 00 53 53 48 5f 54 54 59 3d 2f 64 65 76 2f
2 2 \0 S S H _ T T Y = / d e v /
>22.SSH_TTY=/dev/<
000130 70 74 73 2f 30 00 54 45 52 4d 3d 78 74 65 72 6d
p t s / 0 \0 T E R M = x t e r m

```

```

>pts/0.TERM=xterm<
000140  00  44  49  53  50  4c  41  59  3d  6c  6f  63  61  6c  68  6f
        \0   D   I   S   P   L   A   Y   =   l   o   c   a   l   h   o
>.DISPLAY=localho<
000150  73  74  3a  31  30  2e  30  00  58  44  47  5f  53  45  53  53
        s   t   :   1   0   .   0   \0   X   D   G   _   S   E   S   S
>st:10.0.XDG_SESS<
000160  49  4f  4e  5f  49  44  3d  35  30  32  34  35  00  58  44  47
        I   O   N   _   I   D   =   5   0   2   4   5   \0   X   D   G
>ION_ID=50245.XDG<
000170  5f  52  55  4e  54  49  4d  45  5f  44  49  52  3d  2f  72  75
        _   R   U   N   T   I   M   E   _   D   I   R   =   /   r   u
>_RUNTIME_DIR=/ru<
000180  6e  2f  75  73  65  72  2f  31  30  30  30  00  4c  41  4e  47
        n   /   u   s   e   r   /   1   0   0   0   \0   L   A   N   G
>n/user/1000.LANG<
000190  3d  7a  68  5f  43  4e  2e  55  54  46  2d  38  00
        =   z   h   _   C   N   .   U   T   F   -   8   \0
                                     >=zh_CN.UTF-
8.<

```

该文件及 linux 下环境变量在系统中的内存情况。如偏移量 0x0f 处，该部分内存其实是 C Style 的字符串。在 windows 下，需要使用特殊工具才能看到指定进程（task）的变量情况。

2.2.3. 配置文件

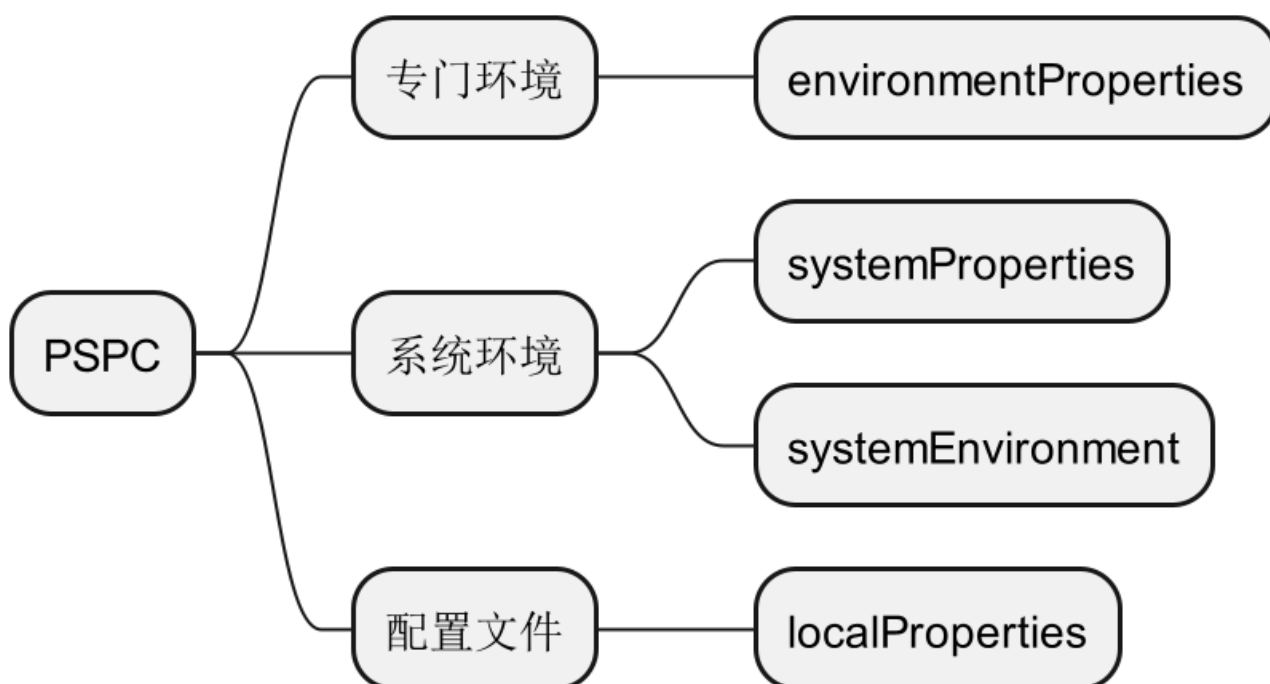
spring 中可以为每个 Context 引入指定的 propertSource。具体引入的方式和规则与各自的情形有关。

3. 具体情形

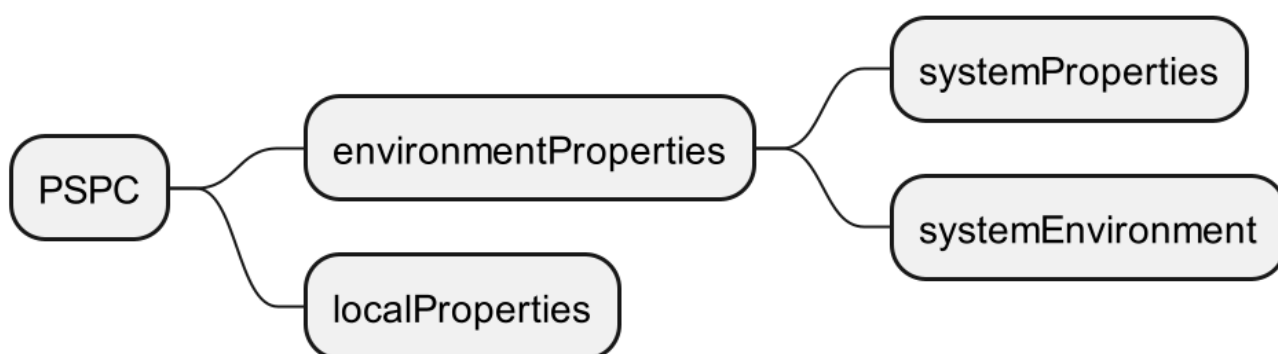
3.1. 纯 spring

理解仅纯 spring 的 environment 变量规则是理解后续更复杂情形的基础。

3.1.1. 一般情形



一般仅集成 spring 时涉及的环境变量源如图，后续将不再区分相关变量源类型。其结构如下图：



再次强调，spring 按照上图中自上而下的顺序获取变量的值，以最先获得的值为准。

3.1.1.1. environmentProperties

是 `org.springframework.context.support.PropertySourcesPlaceholderConfigurer` 的一个实例，但并不实际包含配置变量，而是通过代理特定的配置源实现。在单纯引入 spring 情形下，仅包括 `systemProperties` 和 `systemEnvironment` 两项。该对象一般无法干预。在父子 context 环境下，只有该 properties 内的内容在父子 context 间继承。

3.1.1.2. systemProperties

是 `org.springframework.core.env.PropertiesPropertySource` 的一个实例，其值取自 `java.lang.System.getProperties` 的返回值。对其干预的关键在于在 context 的 environment 生成实例之前调用 `java.lang.System.setProperties` 或 `java.lang.System.setProperty`。

一般的修改方法是通过 JVM 选项修改。如：

```
java -Djdbc_user=demo -jar xxx.jar
```

如上命令就可以将 `jdbc_user` 的值修改为 `demo`。由于 JVM 选项独立于 spring 存在，所以有大量的其他框架也通过该方式提供变化。

3.1.1.3. systemEnvironment

是 `org.springframework.core.env.SystemEnvironmentPropertySource` 的实例，其值来源于系统的环境变量。各种配置与修改的方法见操作系统环境变量的配置方法。特别需要注意的是当应用以 `init.d` 或 `systemd` 的直接子进程方式运行时，环境变量需要特殊配置。

与其他变量源不同，`SystemEnvironmentPropertySource` 获取变量时具有很强的鲁棒性。若取名为 `spring.datasource.jdbc-driver-class` 则若不考虑变量名本身的合法性，环境变量中可以配置如下变量：

```
# 不变
spring.datasource.jdbc-driver-class=xxx
# '.' 替换成 '_'
spring_datasource_jdbc-driver-class=xxx
# '-' 替换成 '_'
spring.datasource.jdbc_driver_class=xxx
# '.' 和 '-' 都替换成 '_'
spring_datasource_jdbc_driver_class=xxx
# 大写
SPRING.DATASOURCE.JDBC-DRIVER-CLASS=xxx
# 大写后 '.' 替换成 '_'
SPRING_DATASOURCE_JDBC-DRIVER-CLASS=xxx
# 大写后 '-' 替换成 '_'
SPRING.DATASOURCE.JDBC_DRIVER_CLASS=xxx
# 大写后 '.' 和 '-' 都替换成 '_'
SPRING_DATASOURCE_JDBC_DRIVER_CLASS=xxx
```

若上述环境变量都配置了，就按照上述顺序取第一个符合条件的配置项。



这里我们建议使用最后一种形式配置,及 `SPRING_DATASOURCE_JDBC_DRIVER_CLASS`.

3.1.1.4. localProperties

也是 `org.springframework.core.env.PropertiesPropertySource` 的一个实例。它通过显式将 properties 引入指定的 context 的方式使用。其方式主要有两种，1. 直接引入键值对儿；2. 引入 properties 文件。

直接将键值对引入由 xml 定义的 context

```
<?xml version="1.0" encoding="utf-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
">
<util:properties id="test">
    <prop key="demoA">cctv</prop>
</util:properties>
<context:property-placeholder properties-ref="test" />
</beans>

```

将 properties 文件或 xml 文件引入由 xml 定义的 context

```

<?xml version="1.0" encoding="utf-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
    ">
    <context:property-placeholder location="classpath:config.properties" />
</beans>

```

事实上，上述两种引入 properties 的方法一般需要复合使用。在默认的配置下，文件中的配置值会覆盖直接配置的变量。



在 spring 5 中除了基于 xml 定义的 context 还有 Java DSL 和 Groovy 相关定义，这里不讨论。另外，spring 自身也提供了诸如 **SimpleCommandLinePropertySource** 等其他实现，但由于默认并未使用将在默认启用的情形下讨论。

xml 的变量配置文件遵守以下声明：

```

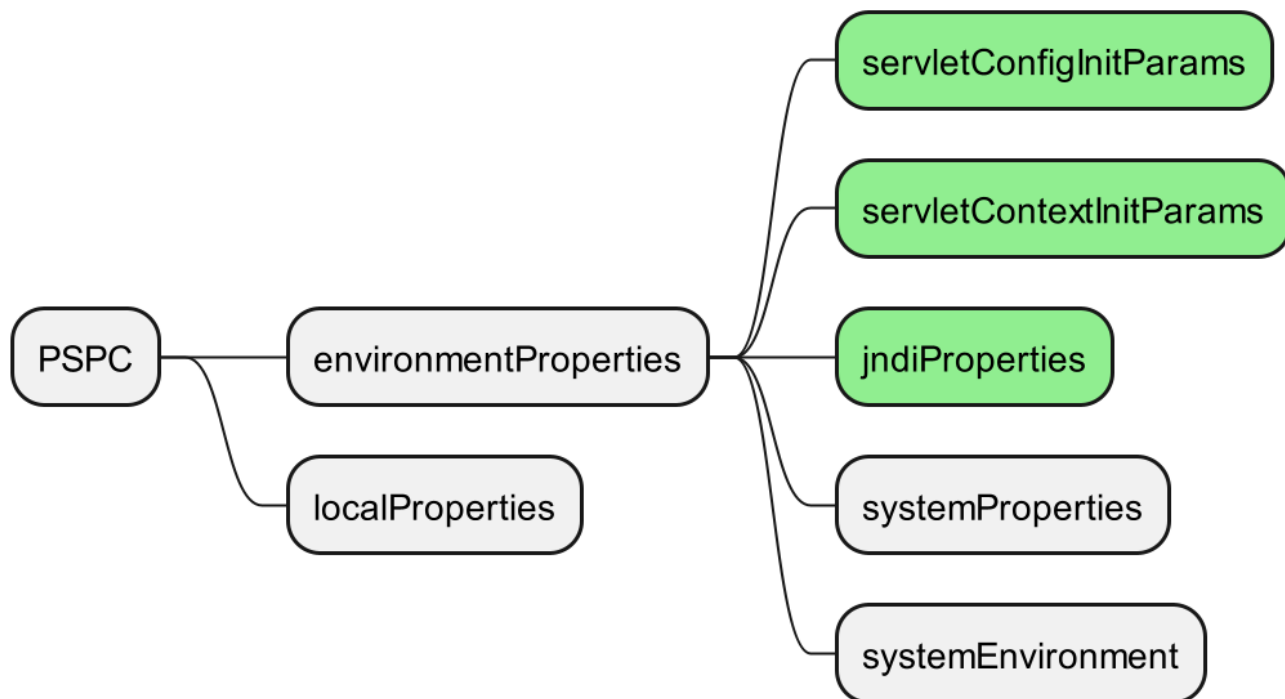
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

```



只有 spring 的条件下，只支持 properties 和 xml 的配置文件，并不支持 yaml 文件。

3.1.2. web 容器内



如上图，绿色部分是 web 容器内相比一般情形增加的部分

3.1.2.1. servletConfigInitParams

`servletConfigInitParams` 是 `org.springframework.web.context.support.ServletConfigPropertySource` 的实例，其仅在使用 `org.springframework.web.servlet.DispatcherServlet` 定义的 `Context` 中可以正常使用，在使用 `org.springframework.web.context.ContextLoaderListener` 定义的 `Context` 中 `servletConfigInitParams` 是一个 `stub` 实现，没有实际功能。

在基于 `web.xml` 配置的 `webApp` 配置如下，具体配置如下方代码中高亮部分：

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <web-app version="3.0"
3     xmlns="http://java.sun.com/xml/ns/javaee"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6     http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/web-
7     app_3_0.xsd">
8     ...
9     <servlet>
10         <servlet-name>mvc</servlet-name>
11         <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
12         class>
```

```

11     <init-param>
12         <param-name>test</param-name>
13         <param-value>servlet</param-value>
14     </init-param>
15 ...
16     <load-on-startup>1</load-on-startup>
17 </servlet>
18 ...
19 </web-app>

```

在基于注解配置的 `webApp` 配置如下，具体配置如下方代码中高亮部分：

```

1 @WebServlet(
2     initParams = {
3         @WebInitParam(
4             name = "test",
5             value = "servlet"
6         )
7     }
8 )

```

3.1.2.2. servletContextInitParams

`servletContextInitParams` 是 `org.springframework.web.context.support.ServletContextPropertySource` 的实例。无论 `org.springframework.web.servlet.DispatcherServlet` 定义的 `Context` 还是 `org.springframework.web.context.ContextLoaderListener` 定义的 `Context` 中都可以使用。

在基于 `web.xml` 配置的 `webApp` 配置如下，具体配置如下方代码中高亮部分：

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <web-app version="3.0"
3     xmlns="http://java.sun.com/xml/ns/javaee"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6     http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/web-
    app_3_0.xsd">
7 ...
8     <context-param>
9         <param-name>test</param-name>
10        <param-value>context</param-value>
11    </context-param>
12 ...
13 </web-app>

```

在基于 **java** 配置的 **webApp** 配置如下，具体配置如下方代码中高亮部分：

```
1 import javax.servlet.ServletContainerInitializer;
2 import javax.servlet.ServletContext;
3 import javax.servlet.ServletException;
4 import java.util.Set;
5
6 public class DemoContainerInitializer implements ServletContainerInitializer {
7     @Override
8     public void onStartUp(Set<Class<?>> c, ServletContext ctx) throws
        ServletException {
9         ctx.setInitParameter("test","context");
10    }
11 }
```

3.1.2.3. jndiProperties

jndiProperties 是 **org.springframework.jndi.JndiPropertySource** 的实例。**jndi** 已经成为 **J2EE** 的标准之一，所有的 **J2EE** 容器都必须提供一个 **JNDI** 的服务。所以该部分的配置跟具体的 **J2EE** 容器有关。

虽然有脱离 **J2EE** 容器的 **jndi** 实现。但是由于 **jndiProperties** 基于 **org.springframework.web.context.support.StandardServletEnvironment** 配置，而其设计用于基于 **Servlet** 的 web 应用 (be used by Servlet-based web applications)。所以除非扩展 **spring** 的 **Environment** 或 **applicationContext**。否则 **jndiProperties** 无法直接在非 web 应用中使用。

这里仅简单介绍几种不同 **J2EE** 或 **Servlet** 容器的配置方式，各种容器不一定完整的实现了所有的 **jndi** 功能的，且实现的复杂程度也不尽相同。其中 Tomcat 和 jetty 这两个 **Servlet** 容器对 **jndi** 的功能实现相对完整，而且其复杂程度与纯 **spring** 环境的变量配置接近。

3.1.2.3.1. tomcat



再次强调，这里 tomcat 中配置 **jndi** 的方式不全，太复杂，不宜完整介绍。

在 **context.xml** 文件中配置方法如下：

```
1 <Context>
2 ...
3     <Environment name="test" value="jndi"
4         type="java.lang.String"/>
5 ...
6 </Context>
```

在 **server.xml** 文件中配置方法如下：

```
1 <Server ...>
```

```

2 ...
3 <GlobalNamingResources ...>
4 ...
5 <Environment name="test" value="jndi"
6     type="java.lang.String"/>
7 ...
8 </GlobalNamingResources>
9 ...
10 </Server>

```

上述两种配置方式再加上一些其他的配置方式，配合覆盖参数与覆盖规则形成一套复杂参数规则。

3.1.2.3.2. jetty



再次强调，这里 jetty 中配置 jndi 的方式不全，太复杂，不宜完整介绍。

jetty 中 jndi 的基本配置方法如下

```

1 <New class="org.eclipse.jetty.plus.jndi.EnvEntry">
2   <Arg></Arg>
3   <Arg>test</Arg>
4   <Arg>jndi</Arg>
5   <Arg type="boolean">true</Arg>
6 </New>

```

根据这一部分出现在 **jetty** 配置文件中的位置不同，出现在不同的配置文件中，再配合 **override** 变量，也能形成复杂的参数规则。

3.2. spring boot 环境

3.2.1. spring boot 环境下相对纯 spring 环境的变更

spring boot 环境下相对纯 spring 环境增加了一些环境变量源之外的变更，其主要集中在上文中的 `localProperties` 部分。

3.2.1.1. yaml 文件支持

如前所述，在纯 spring 环境中是不支持 yaml 配置文件。但从 spring-boot 开始支持 yaml 配置文件。见《Spring Boot Reference Documentation》- 4.2.5. Working with YAML。

虽然 yaml 是一种树形结构但依旧但依旧应该按照键值对进行理解,其对应关系如下：

```

1 environments:
2   dev:
3     url: https://dev.example.com

```

```
4     name: Developer Setup
5   prod:
6     url: https://another.example.com
7     name: My Cool App
```

对应的

```
1 environments.dev.url=https://dev.example.com
2 environments.dev.name=Developer Setup
3 environments.prod.url=https://another.example.com
4 environments.prod.name=My Cool App
```

对于数组，其映射关系如下：

```
1 my:
2   servers:
3     - dev.example.com
4     - another.example.com
```

对应的

```
1 my.servers[0]=dev.example.com
2 my.servers[1]=another.example.com
```

3.2.1.2. 可选的配置文件引入

spring-boot 增加了 optional schema，引入配置文件时用该 schema 修饰。若被修饰的 url 文件资源不存在，也不会导致报错。

3.2.1.3. 支持通配符的配置文件引入

spring-boot 引入文件系统中的配置文件时，其最后一级文件夹可以使用 '*' 通配符。

3.2.1.4. 支持缺少扩展名的文件载入

如前所述，localProperties 通过扩展名确定文件的格式。spring-boot 中，可以使用如下方式载入 yaml 格式的非扩展名文件

```
config[.yaml]
```

3.2.1.5. 支持文件系统式的复合配置

假设文件 '/etc/config/app/test' 的内容如下

```
hello world
```

当 `spring.config.import=optional:configtree:/etc/config` 时相当与配置如下 properties

```
1 app.test=hello world
```

及 spring-boot 引入了 configtree schema 将路径映射为配置变量名，将内容映射为配置变量值。

3.2.1.6. 配置变量中可以引入配置变量

在配置文件中，可以使用 `${param}` 的方式引用其他的配置变量，比如：

```
1 app.name=MyApp
2 app.description=${app.name} is a Spring Boot application
```

此时 `app.description` 的值为 "MyApp is a Spring Boot application"。

3.2.1.7. 基于单文件的多环境配置

在 properties 文件中使用 `#---`，在 yaml 文件中使用 `---` 将单一配置文件分隔成若干个 逻辑文件。如果该逻辑文件中包含 `spring.config.activate.on-cloud-platform` 或 `spring.config.activate.on-profile` 的值，则该逻辑文件在对应的 环境 或 平台 起作用。否则作为公共配置起作用。

3.2.1.8. 宽松绑定

序号	名称	例子	说明
1	kebab case	acme.my-project.person.first-name	以 . 分节，以 - 分隔节内单词
2	camel case	acme.myProject.person.firstName	以 . 分节，小骆驼命名法区分节内单词
3	Underscore notation	acme.my_project.person.first_name	以 . 分节，以 _ 分隔节内单词
4	Upper case format	ACME_MYPROJECT_PERSON_FIRSTNAME	以 _ 分节，单词大写，无法区分节内单词

3.2.1.9. 配置随机值

spring-boot 提供给参数配置随机值的方法。形式如下：

```
1 my:
2   secret: "${random.value}"
3   number: "${random.int}"
4   bignumber: "${random.long}"
5   uuid: "${random.uuid}"
```



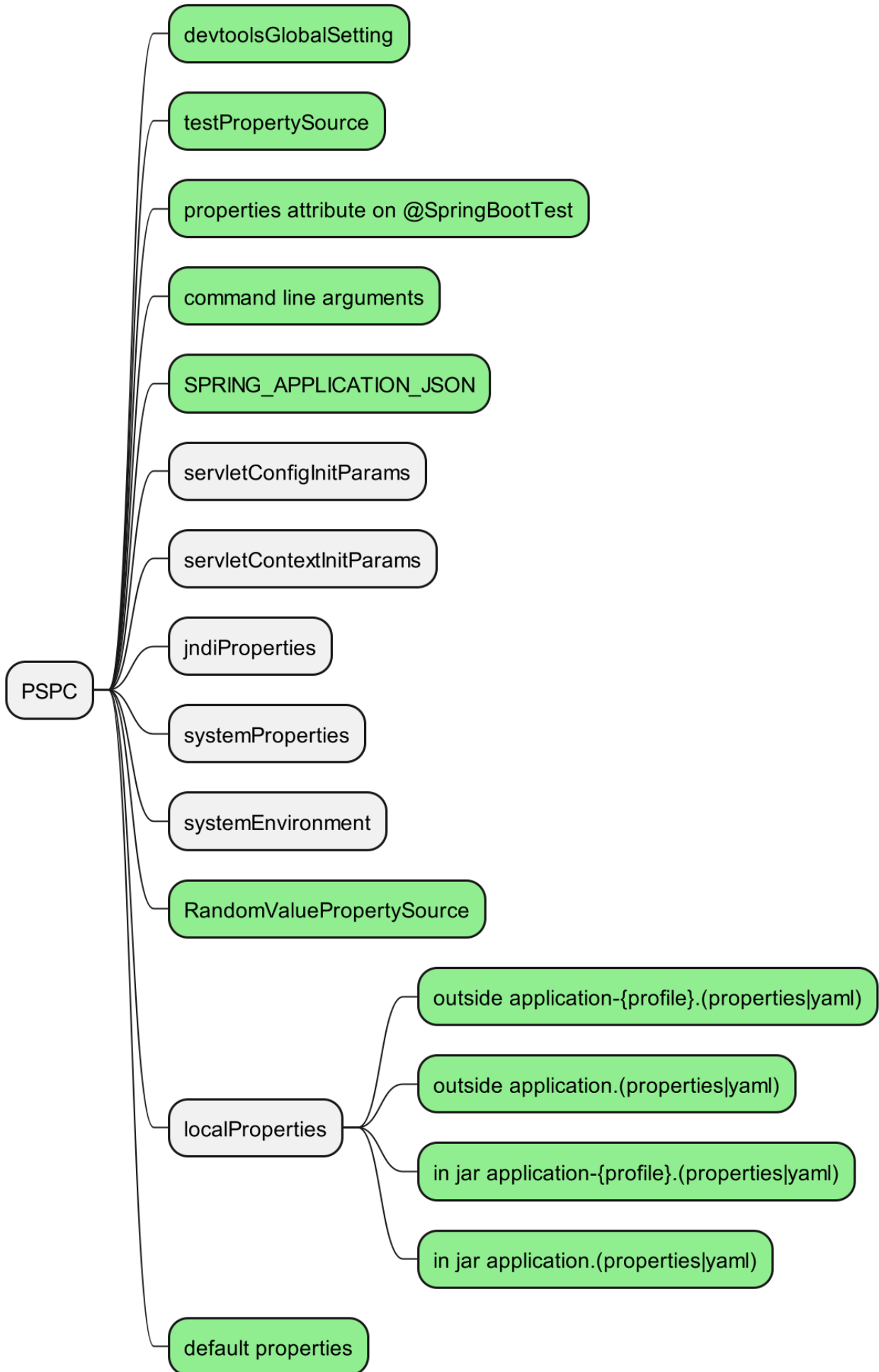
```
6 number-less-than-ten: "${random.int(10)}"  
7 number-in-range: "${random.int[1024,65536]}"
```

3.2.1.10. 配置变量与 bean 绑定

除本文第一节介绍的配置变量引入方式外，spring-boot 提供了一种将一组配置变量绑定为一个专门的 bean 的方式，被称为 **Type-safe Configuration Properties**。

相对于 @Value 比 **Type-safe Configuration Properties** 支持更加宽松的变量绑定模式（@Value 中 "kebab-case using only lowercase letters" 可以绑定 "Camel case" 和 "Upper case format with underscore as the delimiter" 模式，反之不行）。

3.2.2. spring boot 的配置源



如图所示，相对于纯粹的 spring 环境 spring-boot 增加了 7 个配置点，此外还扩展和预设了原 spring 中的 localProperties。

[spring boot property sources] | figure/spring-boot-property-sources.jpg

图 3. 典型的配置栈

如上图所示，在 spring-boot 中不再由 `PropertySourcesPropertyResolver` 提供参数解析，而是由 `org.springframework.boot.context.properties.source.ConfigurationPropertySourcesPropertyResolver` 提供对应的功能。同时从上图中可以看出，有一些 propertySource 并未在思维导图中列出。未列出的我们无法控制，不在本文的讨论中。以下是其结构的字符串形式：

```
[
  MapPropertySource {name='server.ports'},
  ConfigurationPropertySourcesPropertySource@1161350493 {
    name='configurationProperties',
    properties=org.springframework.boot.context.properties.source.SpringConfigurationPropertySources@55ae625
  },
  OriginTrackedMapPropertySource@1150795225 {
    name='devtools-local: [file:/C:/Users/wangj/.config/spring-boot/spring-boot-devtools.properties]',
    properties={test=prop}
  },
  OriginTrackedMapPropertySource@1121337133 {
    name='devtools-local: [file:/C:/Users/wangj/.config/spring-boot/spring-boot-devtools.yaml]',
    properties={test=yaml}
  },
  OriginTrackedMapPropertySource@1090994191 {
    name='devtools-local: [file:/C:/Users/wangj/.config/spring-boot/spring-boot-devtools.yml]',
    properties={test=yml}
  },
  StubPropertySource {name='servletConfigInitParams'},
  ServletContextPropertySource@1752743208 {
    name='servletContextInitParams',
    properties=org.apache.catalina.core.ApplicationContextFacade@6e090721
  },
  PropertiesPropertySource {name='systemProperties'},
  OriginAwareSystemEnvironmentPropertySource@331283908 {
    name='systemEnvironment',
    properties={
      PROCESSOR_LEVEL=6,
      ...
    }
  }
]
```

```

    }
  },
  RandomValuePropertySource@1782510367 {name='random',
properties=java.util.Random@19e29622},
  OriginTrackedMapPropertySource@1646176809 {
    name='Config resource 'class path resource [application.yml]' via location
'optional:classpath:/',
    properties={
      ecloude.dateFormat=yyyy-MM-dd HH:mm:ss.SSS,
      ...
    }
  },
  MapPropertySource {name='devtools'},
  @2131007858 {name='Management Server', properties=java.lang.Object@b29849a}
]

```

3.2.2.1. DevtoolsGlobalSetting

spring-boot-devtools 是 spring-boot 提供的开发时工具包。通过以下方法引入 spring-boot app 工程模块 (module) 后启用 **DevtoolsGlobalSetting**:

```

<dependencies>
...
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
</dependency>
...
</dependencies>

```

```

dependencies {
  developmentOnly("org.springframework.boot:spring-boot-devtools")
}

```

此时，在 **\$HOME/.config/spring-boot** 依次在如下文件中查找配置变量：

1. spring-boot-devtools.properties
2. spring-boot-devtools.yaml
3. spring-boot-devtools.yml

此外，还会生成一组名为 **devtools** 优先级非常低的配置，相关配置如下：

```

Map<String, Object> properties = new HashMap<>();

```

```

properties.put("spring.thymeleaf.cache", "false");
properties.put("spring.freemarker.cache", "false");
properties.put("spring.groovy.template.cache", "false");
properties.put("spring.mustache.cache", "false");
properties.put("server.servlet.session.persistent", "true");
properties.put("spring.h2.console.enabled", "true");
properties.put("spring.web.resources.cache.period", "0");
properties.put("spring.web.resources.chain.cache", "false");
properties.put("spring.template.provider.cache", "false");
properties.put("spring.mvc.log-resolved-exception", "true");
properties.put("server.error.include-binding-errors", "ALWAYS");
properties.put("server.error.include-message", "ALWAYS");
properties.put("server.error.include-stacktrace", "ALWAYS");
properties.put("server.servlet.jsp.init-parameters.development", "true");
properties.put("spring.reactor.debug", "true");

```

对于 `DevtoolsGlobalSetting` 的使用应注意以下问题：

1. DevtoolsGlobalSetting 是一组仅应用于开发环境的配置，优先级极高。
2. DevtoolsGlobalSetting 不对应单一的 spring boot application，而是对应运行在该开发环境下所有的 spring boot application，所以不应用于生产。
3. 只有引用了 spring-boot-devtools 依赖后才可以使使用。
4. DevtoolsGlobalSetting 不存在 profile 机制。
5. DevtoolsGlobalSetting 不是单一文件，而是对应三种文件，有序。
6. Devtools 会额外生成一组低优先级的默认配置，不可干预。

3.2.2.2. testPropertySource

testPropertySource 是指使用 `@TestPropertySource` 提供的配置源。主要用法如下：

```

@ContextConfiguration
@TestPropertySource(locations={"/test.properties"})
class MyIntegrationTests {
    // class body...
}

```

```

@ContextConfiguration
@TestPropertySource(properties = {
    "timezone = GMT",
    "port: 4242",
    "test 123456"
})

```

```
class MyIntegrationTests {  
    // class body...  
}
```

如上代码，`@TestPropertySource` 主要有 `locations` 和 `properties` 两个属性的用法，其中前者表示配置文件路径，后者表示配置文件内容。`properties` 支持 `=:` 和空格三种分隔符。此外 `locations` 和 `value` 属性互为别名。

`@TestPropertySource` 并非是 spring-boot 提供的注解，在 spring-test 中就给出用法，但由于其仅用于单元测试场景，所以很少提及。

总之，`@TestPropertySource` 是一种仅用于单元测试的配置源，仅在开发环境和 CI 环境起作用。

3.2.2.3. properties attribute on @SpringBootTest

`@SpringBootTest` 是 spring boot 提供的开箱即用的单元测试注解。其中 `properties` 属性提供类似 `@TestPropertySource` 的 `properties` 的用法。

```
@SpringBootTest(properties = "spring.main.web-application-type=reactive")  
class MyWebFluxTests {  
    ...  
}
```

如上所述，`@SpringBootTest` 的 `properties` 属性与 `@TestPropertySource` 的 `properties` 属性完全相同。只是其作用点不同。`@SpringBootTest` 仅用于单元测试，也就是开发和 CI 环境。

3.2.2.4. command line arguments

3.3. spring cloud 环境

4. 其他

5. 作业