
Абдульманов Роман @NevRA
Пользователь

6 апреля 2011 в 00:33

Разбор XML при помощи Simple Framework

Разработка под Android*

Simple
XML SERIALIZATION

Здравствуйте, читатели Хабрахабр!

Данный пост навеян другим постом и комментарием уважаемого хабраюзера @AnatolyB оттуда.

Я думаю, что многие знакомы с данной библиотекой, но, тем не менее, оказалось, что она еще не была отражена на страницах Хабры. Исправлением этого недоразумения мы и займемся сегодня.

И, конечно же, тем, кто еще не знаком с этой прекрасной библиотекой, рекомендую скорее познакомиться, я же постараюсь в этом вам помо

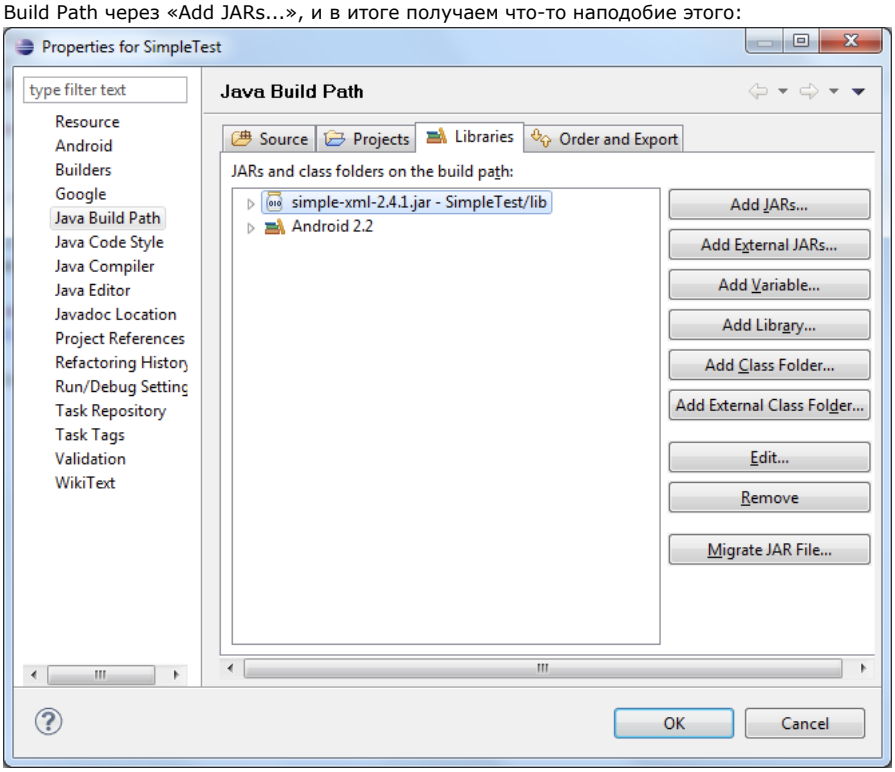
Intro

Simple Framework — это библиотека, совместимая с виртуальной машиной Android, предназначенная для сериализации/десериализации об Java в xml и обратно.

Основным достоинством Simple является декларативный подход к описанию связей между классами с их содержимым и XML представлением достаточно задать соответствующие атрибуты полям класса и можно тут же их сериализовать в XML, а при желании и обратно. Никакого дополнительного мусора, вроде перечисления нод, делать не нужно, все очень просто и прозрачно. Всю работу с сопоставления полей класс перечислением, получением значений и т.п. берет на себя Simple, а помогает ей в этом reflection.

На официальном сайте имеется все нужное для начала работы с библиотекой, а так же мой любимый вид документации: документация с бол набором разносторонних примеров.

Первым делом качаем саму библиотеку отсюда. Находим jar файл и копируем его в папку lib внутри нашего проекта. Далее добавляем наш ja Build Path через «Add JARs...», и в итоге получаем что-то наподобие этого:



Примеры использования

Теперь мы можем использовать Simple в нашем проекте. И первое что мы попытаемся сделать это преобразуем простой XML в class Java.

Пусть у нас есть следующий XML:

```
<Pet>
   <Name>Bobby</Name>
   <Age>8</Age>
   <NickName>Lucky</NickName>
</Pet>
```

Он представляет сущность «домашнее животное», у которой есть обязательные поля: имя и возраст, а также одно необязательное: кличка.

Для того, чтобы связать этот XML с каким-то классом нам нужно создать класс и добавить к нему специальные атрибуты, чтобы Simple смог г чего и как мы хотим сделать. Для нашего примера класс будет выглядеть вот так:

```
@Root(name="Pet")
public class MyPet
{
    @Element(name="Name")
    public String name;

    @Element(name="Age")
    public int age;

    @Element(required=false, name="NickName")
    public String nickName;
}
```

Где:

- **Root** — указывает на root'ый элемент XML.
- **Element** — внутренние поля «Pet».
- **required** — говорит, что это поле может быть опциональным.
- **name** — указывает на имя элемента во входном XML, если это имя и поле класса совпадают, то данный атрибут может быть опущен.

Если поля Pet задавались бы атрибутами, а не элементами, то входной XML выглядел бы так:

```
<Pet Name="Bobby" Age="8" NickName="Lucky"/>
```

А наш класс выглядел бы вот так:

```
@Root(name="Pet")
public class MyPet
{
    @Attribute(name="Name")
    public String name;

    @Attribute(name="Age")
    public int age;

    @Attribute(required=false, name="NickName")
    public String nickName;
}
```

Сам код по десериализации выглядит так:

```
Reader reader = new StringReader(xml);
Persister serializer = new Persister();
try
{
    MyPet pet = serializer.read(MyPet.class, reader, false);
    Log.v("SimpleTest", "Pet Name" + pet.name);
}
catch (Exception e)
{
    Log.e("SimpleTest", e.getMessage());
}
```

В приведенном выше коде мы отдали на вход Simple serializer два параметра: MyPet.class — указание на класс с описанием атрибутов для десериализации и reader — поток содержащий входную XML. Как видно код совсем не сложный и очень компактный.

Код для обратного преобразования так же достаточно простой:

```
Writer writer = new StringWriter();
Serializer serializer = new Persister();
try
{
    MyPet pet = new MyPet();
    pet.name = "Bobby";
    pet.age = 8;
    pet.nickName = "Lucky";

    serializer.write(pet, writer);
    String xml = writer.toString();
}
catch (Exception e)
{
    Log.e("SimpleTest", e.getMessage());
}
```

Для соблюдения принципа инкапсуляции, поля класса можно обернуть в get'еры и set'еры и Simple будет работать с ними:

```
@Root(name="Pet")
public class MyPet
{
    private String name;
    private int age;
    private String nickName;

    @Attribute(name="Name")
    public void setName(String name)
    {
        this.name = name;
    }

    @Attribute(name="Name")
    public String getName()
    {
        return name;
    }

    @Attribute(name="Age")
    public void setAge(int age)
    {
        this.age = age;
    }

    @Attribute(name="Age")
    public int getAge()
    {
        return age;
    }

    @Attribute(required=false, name="NickName")
    public void setNickName(String nickName)
    {
        this.nickName = nickName;
    }

    @Attribute(required=false, name="NickName")
    public String getNickName()
    {
        return nickName;
    }
}
```

Если элемент XML имеет свои атрибуты или вложенные элементы, то его можно объявить отдельным классом или их списком. Модифицируем пример, добавим сущность «питомник» (nursery), которая может содержать произвольное число объектов «домашнее животное» (Pet). Приме

```
<Nursery>
    <Pet Name="Bobby" Age="8" NickName="Lucky"/>
```

```
    <Pet Name="Rex" Age="3"/>
    <Pet Name="Pumba" Age="1"/>
</Nursery>
```

Для этого примера единственное, что нам нужно, это добавить класс для «Nursery»:

```
@Root(name="Nursery")
public class MyNursery
{
    @ElementList(inline=true, name="Pet")
    public List<MyPet> pets;
}
```

Как видно, все так же просто. Ключевое слово **inline** говорит о том, что элементы «Pet» содержаться сразу внутри MyNursery, без использова промежуточного родительского элемента.

Код для загрузки «Nursery» аналогичен тому, что мы делали для «Pet»:

```
Reader reader = new StringReader(xml);
Persister serializer = new Persister();
try
{
    MyNursery nursery = serializer.read(MyNursery.class, reader, false);
    Log.v("SimpleTest", "Pets in nursery: " + nursery.pets.size());
}
catch (Exception e)
{
    Log.e("SimpleTest", e.getMessage());
}
```

Я думаю этих примеров достаточно для начала самостоятельного разбирательство с библиотекой Simple. Тем более, что большое количество примеров представлено на официальном сайте.

Заключение

Simple предлагает решения, наверно, для всех конструкций, которые только возможны в XML. Так же поддерживаются такие возможности яз Java как: наследование и интерфейсы.

При использовании Simple можно отметить следующие положительные и отрицательные стороны.

Положительные:

1. Простота в использовании и понимании.
2. Количество кода при таком подходе минимально.
3. Поддержка платформы Android.
4. Богатые возможности по поддержке различных конструкций XML.
5. Возможность применять в не Android приложениях, например в отвязанных от устройств Unit-тестах.
6. Лицензия Apache, т.е. можно свободно использовать в коммерческом софте.

Отрицательные:

1. Синтаксис Simple атрибутов перегружает представление классов.
2. Для работы Simple применяется механизм Reflection'a, а это затратные операции. Поэтому, если вы собираетесь применять данный Fran в продукте требовательном к производительности, то стоит задуматься над целесообразностью такого решения.

Полезные ссылки

1. Официальный сайт.
2. Прямая ссылка на документацию.
3. Другая статья на тему.

Simple Framework, Java, Android, XML, serialization

16 мая 2015 в 14:59
Обход CloudFlare ScrapeShield в Java (Android)
+16 12,1k 53 5

14 января 2014 в 16:58
Новый вид разработчиков — Framework Java Coder?
+65 44,8k 104 209

11 ноября 2012 в 00:50
Text Mining Framework (Java)
+32 23,5k 133 39

САМОЕ ЧИТАЕМОЕ Разраб

Сейчас | Сутки | Неделя | Месяц

Получил 1.2K звезд на GitHub с ужасной архитектурой. Как?
+22 36,4k 89 36

Реализация псевдо-3D в гоночных играх
+90 13k 172 16

Индексы в PostgreSQL — 1
+102 11,5k 303 32

Выбор MQ для высоконагруженного проекта
+30 9,9k 131 47

Алгоритм Джонкера-Волгенанта + t-SNE = супер-сила
+63 9,5k 146 2

Комментарии (13)

Bektimirov 6 апреля 2011 в 09:11

Аннотации хороши, когда их мало. При сложной структуре XML документа код обрастает и читаемость его сильно ухудшается. Кстати, JAXB ведь делает то же Почему не использовать его?

> **NevRA** 6 апреля 2011 в 09:46
>
> >Кстати, JAXB ведь делает то же самое. Почему не использовать его?
>
> JAXB не работает на виртуальной машине Android, по крайней мере, так было еще совсем недавно.

pilgr 6 апреля 2011 в 09:40

Спасибо, нужно будет попробовать ее в одном из своих **Pet**-проектов

Fury 6 апреля 2011 в 10:16

Как раз недавно тоже задавался вопросом, что есть под Android для Xml биндинга, и тоже посмотрел на эту библиотеку. Но даже не стал её пробовать из-за её размера.
Можете назвать меня старомодным (тяга к сокращению размера приложения осталась со времён J2ME:))), но добавлять к проекту либу размером в 300кб толь того, чтобы вам было удобнее парсить XML, мне кажется каким-то неуважением к своим пользователям.
Так что, как минимум, стоит добавить к минусам размер библиотеки.

Fury 6 апреля 2011 в 10:38

Интересно было бы выслушать мнение минусующих :)

NevRA 6 апреля 2011 в 11:18

Я не минусовал, но мнение выскажу.
Компилятор ее ужмет при добавлении в итоговый apk, т.е. 300K библиотеки не значит что размер итогового пакета получится 300K + остальное.
Да и общая тенденция, что уже особенно никто не следит за размерами такого порядка, у людей высокоскоростной интернет и флэшки на гигабайты.
Но конечно нужно стремится к уменьшению итогового размеров пакета, в разумных пределах, калькулятор в 20 метров это все же не дело. ;)

Fury 6 апреля 2011 в 12:47

> Компилятор ее ужмет при добавлении в итоговый apk

Да, пожалуй, стоит проверить. Но не думаю, что выигрыш будет больше, чем даёт повторная переупаковка, то есть ужатие до ~270кб. Ещё дополнительн выигрыш может дать обфускация, но неизвестно, не отвалится ли там чего из-за использования рефлекшена :) В любом случае, будет больше 200кб ове И очень печально, что это не считается минусом.

> Да и общая тенденция, что уже особенно никто не следит за размерами такого порядка, у людей высокоскоростной интернет и флэшки на гигабайты.

Учитывая статистику распределения версий, как минимум, тридцать процентов пользователей (те, у кого версия Андроид ниже 2.2) ещё, точно, следят. Д я слежу, хотя у меня 2.3.3, так как даже при инсталляции на флэшку часть файлов ставится в основную память (кстати, давно было интересно, что же ос основной памяти).

В общем, повторюсь — я не против самой либы (на серверной стороне сам пользуюсь аналогами без каких-либо раздумий о размере), но в контексте разработки для мобильных платформ размер всё-таки нужно учитывать, и очень хотелось бы, чтобы разработчики это понимали.

ivanrt 6 апреля 2011 в 15:51

В основную память копируются распакованные далвик классы, в dalvikCache. Я пока писал свой diskusage намучился пока просёк что как считается.

ArtRoman 8 апреля 2011 в 01:57

Не так давно я сделал соответствующий пост в бложик, буду рад, если это объяснит. Андроид 2.3 тут сильно выигрывает более эффективным пере (странно, что не реализовали это сразу).

Aivean 6 апреля 2011 в 11:29

Скажите, а чем это лучше/хуже XStream?

NevRA 6 апреля 2011 в 11:46

Не работал с XStream, поэтому не могу вам ответить.
Но выглядит очень похоже на Simple, смущает только, что данная библиотека не обновлялась с 2008 года, а это как-то сразу отталкивает.

gshock 6 апреля 2011 в 12:23

>T.e., как мы привыкли в .NET
Ну, не все привыкли в .NET работать :)
А за обзор спасибо. Возьму на заметку

NevRA 6 апреля 2011 в 12:25

Да, вы правы. Исправил.

Scilab в свободном падении

777 5 5

Электроника для ролевых игр в 2016 году GT

2,2k 5 9

Тысячная избранная статья. Как устроено рецензирование в Википедии GT

2,8k 13 31

Библиотека компонентов как инструмент поддержания целостности вебсайта

1,6k 13 6