个人资料

程序员的自我反思

关注 发私信

访问： 37273次
积分： 701
等级： BLOG 3
排名： 千里之外

原创： 30篇 转载： 0篇
译文： 0篇 评论： 22条

Android View 深度分析requestLayout、invalidate与postInvalidate

标签： android View invalidate 布局 源码

2016-06-04 10:03 3865人阅读 评论(1) 收藏 举报

分类： Android（27）

前言

前几篇文章中，笔者对View的三大工作流程进行了详细分析，而这篇文章则详细讲述与三大工作流程密切相关的两个方法，分别是requestLayout和invalidate，如果对Viwe的三个工作流程不熟悉的读者，可以先看看前几篇文章，以便能更容易理解这篇文章的内容。

requestLayout

当我们动态移动一个View的位置，或者View的大小、形状发生了变化的时候，我们可以在view中调用这个方法，即：

```
view.requestLayout();
```

那么该方法的作用是什么呢？

从方法名字可以知道，“请求布局”，那就是说，如果调用了这个方法，那么对于一个子View来说，应该会重新进行布局流程。但是，真实情况略有不同如果子View调用了这个方法，其实会从View树重新进行一次测量、布局、绘制这三个流程，最终就会显示子View的最终情况。那么，这个方法是怎么实的呢？我们从源码角度进行解析。

首先，我们看**View#requestLayout**方法：

```
/**
 * Call this when something has changed which has invalidated the
 * layout of this view. This will schedule a layout pass of the view
 * tree. This should not be called while the view hierarchy is currently in a layout
 * pass ({@link #isInLayout()}. If layout is happening, the request may be honored at the
 * end of the current layout pass (and then layout will run again) or after the current
 * frame is drawn and the next layout occurs.
 *
 * <p>Subclasses which override this method should call the superclass method to
 * handle possible request-during-layout errors correctly.</p>
 */
//从源码注释可以看出，如果当前View在请求布局的时候，View树正在进行布局流程的话，
//该请求会延迟到布局流程完成后或者绘制流程完成且下一次布局发现的时候再执行。
@CallSuper
public void requestLayout() {
    if (mMeasureCache != null) mMeasureCache.clear();

    if (mAttachInfo != null && mAttachInfo.mViewRequestingLayout == null) {
        // Only trigger request-during-layout logic if this is the view requesting it,
        // not the views in its parent hierarchy
        ViewRootImpl viewRoot = getViewRootImpl();
        if (viewRoot != null && viewRoot.isInLayout()) {
            if (!viewRoot.requestLayoutDuringLayout(this)) {
                return;
            }
        }
        mAttachInfo.mViewRequestingLayout = this;
    }

    //为当前view设置标记位 PFLAG_FORCE_LAYOUT
    mPrivateFlags |= PFLAG_FORCE_LAYOUT;
    mPrivateFlags |= PFLAG_INVALIDATED;

    if (mParent != null && !mParent.isLayoutRequested()) {
        //向父容器请求布局
        mParent.requestLayout();
    }
```

Android View 深度分析request… (1)
Yii 2.0学习日记：用户登陆详… (0)

推荐文章

*Android官方开发文档Training系列课程中文版：网络操作之XML解析
*Delta - 轻量级JavaWeb框架使用文档
*Nginx正反向代理、负载均衡等功能实现配置
* 浅析ZeroMQ工作原理及其特点
*android源码解析（十九）-->Dialog加载绘制流程
*Spring Boot 实践折腾记（三）：三板斧，Spring Boot下使用Mybatis

最新评论

Android View 深度分析requestLayout、inv…
长城Great ：牛

揭开RecyclerView的神秘面纱(二)：处理R…
长城Great ：真心不错！

Android View源码解读：浅谈DecorView…
程序员的自我反思 ：@xiehuimx:因为有些类是隐藏的，加了@hide标记，所以不会提示。

Android View源码解读：浅谈DecorView…
xiehuimx ：eclipse关联sdk目录sources下的源码，为什么有的类（例如PhoneWindow）不提示...

Android IPC机制(二):AIDL的基本使用方法
dingyongby ：非常简单易懂，一目了然。能让你知道可以做什么，网上都是说 原理怎么怎么样，也没说 实际使用场景。为了...

Android IPC机制(二):AIDL的基本使用方法
dingyongby ：非常简单易懂，一目了然。能让你知道可以做什么，网上都是说 原理怎么怎么样，也没说 实际使用场景。为了...

Android View 测量流程(Measure)完全解析
AnalyzeSystem ：不错，费心了

Android View 测量流程(Measure)完全解析
程序员的自我反思 ：@csdnYF:嗯，谢谢支持

Android ToolBar 使用完全解析
yjmAndroid ：谢谢 已解决

Android ToolBar 使用完全解析
程序员的自我反思 ：@yjmAndroid:在build.gradle中添加如下依赖：compile 'com.andr...

```
38      if (mAttachInfo != null && mAttachInfo.mViewRequestingLayout == this) {
39          mAttachInfo.mViewRequestingLayout = null;
40      }
41  }
```

在requestLayout方法中，首先先判断当前View树是否正在布局流程，接着为当前子View设置标记位，该标记位的作用就是标记了当前的View是需要进行新布局的，接着调用mParent.requestLayout方法，这个十分重要，因为这里是向父容器请求布局，即调用父容器的requestLayout方法，为父容器添加PFLAG_FORCE_LAYOUT标记位，而父容器又会调用它的父容器的requestLayout方法，即requestLayout事件层层向上传递，直到DecorView，即根View，而根View又会传递给ViewRootImpl，也即是说子View的requestLayout事件，最终会被ViewRootImpl接收并得到处理。纵观这个向上传递的流程，其实是采用了责任链模式，即不断向上传递该事件，直到找到能处理该事件的上级，在这里，只有ViewRootImpl能够处理requestLayout事件。

在ViewRootImpl中，重写了requestLayout方法，我们看看这个方法，**ViewRootImpl#requestLayout**:

```
1   @Override
2   public void requestLayout() {
3       if (!mHandlingLayoutInLayoutRequest) {
4           checkThread();
5           mLayoutRequested = true;
6           scheduleTraversals();
7       }
8   }
```

在这里，调用了scheduleTraversals方法，这个方法是一个异步方法，最终会调用到**ViewRootImpl#performTraversals**方法，这也是View工作流程的核心方法，在这个方法内部，分别调用measure、layout、draw方法来进行View的三大工作流程，对于三大工作流程，前几篇文章已经详细讲述了，这里再做一点补充说明。

先看**View#measure**方法：

```
1   public final void measure(int widthMeasureSpec, int heightMeasureSpec) {
2        ...
3
4       if ((mPrivateFlags & PFLAG_FORCE_LAYOUT) == PFLAG_FORCE_LAYOUT ||
5               widthMeasureSpec != mOldWidthMeasureSpec ||
6               heightMeasureSpec != mOldHeightMeasureSpec) {
7           ...
8           if (cacheIndex < 0 || sIgnoreMeasureCache) {
9               // measure ourselves, this should set the measured dimension flag back
10              onMeasure(widthMeasureSpec, heightMeasureSpec);
11              mPrivateFlags3 &= ~PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT;
12          }
13          ...
14          mPrivateFlags |= PFLAG_LAYOUT_REQUIRED;
15      }
16  }
```

首先是判断一下标记位，如果**当前View的标记位为PFLAG_FORCE_LAYOUT**，那么就会进行测量流程，调用onMeasure，对该View进行测量，接着最后为标记位设置为**PFLAG_LAYOUT_REQUIRED**,这个标记位的作用就是在View的layout流程中，如果当前View设置了该标记位，则会进行布局流程。具体可以看如下**View#layout**源码：

```
1   public void layout(int l, int t, int r, int b) {
2       ...
3       //判断标记位是否为PFLAG_LAYOUT_REQUIRED，如果有，则对该View进行布局
4       if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) == PFLAG_LAYOUT_REQUIRED) {
5           onLayout(changed, l, t, r, b);
6           //onLayout方法完成后，清除PFLAG_LAYOUT_REQUIRED标记位
7           mPrivateFlags &= ~PFLAG_LAYOUT_REQUIRED;
8           ListenerInfo li = mListenerInfo;
9           if (li != null && li.mOnLayoutChangeListeners != null) {
10              ArrayList<OnLayoutChangeListener> listenersCopy =
11                      (ArrayList<OnLayoutChangeListener>)li.mOnLayoutChangeListeners.clone();
12              int numListeners = listenersCopy.size();
13              for (int i = 0; i < numListeners; ++i) {
14                  listenersCopy.get(i).onLayoutChange(this, l, t, r, b, oldL, oldT, oldR, oldB);
15              }
16          }
17      }
18
19      //最后清除PFLAG_FORCE_LAYOUT标记位
20      mPrivateFlags &= ~PFLAG_FORCE_LAYOUT;
21      mPrivateFlags3 |= PFLAG3_IS_LAID_OUT;
22  }
```

那么到目前为止，requestLayout的流程便完成了。

小结：子View调用requestLayout方法，会标记当前View及父容器，同时逐层向上提交，直到ViewRootImpl处理该事件，ViewRootImpl会调用三大流程，从measure开始，对于每一个含有标记位的view及其子View都会进行测量、布局、绘制。

invalidate

该方法的调用会引起View树的重绘，常用于内部调用(比如 setVisiblity())或者需要刷新界面的时候,需要在主线程(即UI线程)中调用该方法。那么我们来分析一下它的实现。

首先，一个子View调用该方法，那么我们直接看**View#invalidate**方法：

```
public void invalidate() {
    invalidate(true);
}
void invalidate(boolean invalidateCache) {
    invalidateInternal(0, 0, mRight - mLeft, mBottom - mTop, invalidateCache, true);
}
void invalidateInternal(int l, int t, int r, int b, boolean invalidateCache,
        boolean fullInvalidate) {
    if (mGhostView != null) {
        mGhostView.invalidate(true);
        return;
    }

    //这里判断该子View是否可见或者是否处于动画中
    if (skipInvalidate()) {
        return;
    }

    //根据View的标记位来判断该子View是否需要重绘，假如View没有任何变化，那么就不需要重绘
    if ((mPrivateFlags & (PFLAG_DRAWN | PFLAG_HAS_BOUNDS)) == (PFLAG_DRAWN | PFLAG_HAS_BOUNDS)
            || (invalidateCache && (mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == PFLAG_DRAWING_CACHE_VA
            || (mPrivateFlags & PFLAG_INVALIDATED) != PFLAG_INVALIDATED
            || (fullInvalidate && isOpaque() != mLastIsOpaque)) {
        if (fullInvalidate) {
            mLastIsOpaque = isOpaque();
            mPrivateFlags &= ~PFLAG_DRAWN;
        }

        //设置PFLAG_DIRTY标记位
        mPrivateFlags |= PFLAG_DIRTY;

        if (invalidateCache) {
            mPrivateFlags |= PFLAG_INVALIDATED;
            mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
        }

        // Propagate the damage rectangle to the parent view.
        //把需要重绘的区域传递给父容器
        final AttachInfo ai = mAttachInfo;
        final ViewParent p = mParent;
        if (p != null && ai != null && l < r && t < b) {
            final Rect damage = ai.mTmpInvalRect;
            damage.set(l, t, r, b);
            //调用父容器的方法，向上传递事件
            p.invalidateChild(this, damage);
        }
        ...
    }
}
```

可以看出，invalidate有多个重载方法，但最终都会调用**invalidateInternal**方法，在这个方法内部，进行了一系列的判断，判断View是否需要重绘，接着为

该View设置标记位，然后把需要重绘的区域传递给父容器，即调用父容器的invalidateChild方法。

接着我们看**ViewGroup#invalidateChild**：

```
/**
 * Don't call or override this method. It is used for the implementation of
 * the view hierarchy.
 */
public final void invalidateChild(View child, final Rect dirty) {

    //设置 parent 等于自身
    ViewParent parent = this;

    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        // If the child is drawing an animation, we want to copy this flag onto
        // ourselves and the parent to make sure the invalidate request goes
        // through
        final boolean drawAnimation = (child.mPrivateFlags & PFLAG_DRAW_ANIMATION)
                == PFLAG_DRAW_ANIMATION;

        // Check whether the child that requests the invalidate is fully opaque
        // Views being animated or transformed are not considered opaque because we may
        // be invalidating their old position and need the parent to paint behind them.
        Matrix childMatrix = child.getMatrix();
        final boolean isOpaque = child.isOpaque() && !drawAnimation &&
                child.getAnimation() == null && childMatrix.isIdentity();
        // Mark the child as dirty, using the appropriate flag
        // Make sure we do not set both flags at the same time
        int opaqueFlag = isOpaque ? PFLAG_DIRTY_OPAQUE : PFLAG_DIRTY;

        if (child.mLayerType != LAYER_TYPE_NONE) {
            mPrivateFlags |= PFLAG_INVALIDATED;
            mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
        }

        //储存子View的mLeft和mTop值
        final int[] location = attachInfo.mInvalidateChildLocation;
        location[CHILD_LEFT_INDEX] = child.mLeft;
        location[CHILD_TOP_INDEX] = child.mTop;

        ...

        do {
```

```
            View view = null;
            if (parent instanceof View) {
                view = (View) parent;
            }

            if (drawAnimation) {
                if (view != null) {
                    view.mPrivateFlags |= PFLAG_DRAW_ANIMATION;
                } else if (parent instanceof ViewRootImpl) {
                    ((ViewRootImpl) parent).mIsAnimating = true;
                }
            }

            // If the parent is dirty opaque or not dirty, mark it dirty with the opaque
            // flag coming from the child that initiated the invalidate
            if (view != null) {
                if ((view.mViewFlags & FADING_EDGE_MASK) != 0 &&
                        view.getSolidColor() == 0) {
                    opaqueFlag = PFLAG_DIRTY;
                }
                if ((view.mPrivateFlags & PFLAG_DIRTY_MASK) != PFLAG_DIRTY) {
                    //对当前View的标记位进行设置
                    view.mPrivateFlags = (view.mPrivateFlags & ~PFLAG_DIRTY_MASK) | opaqueFlag;
                }
            }

            //调用ViewGrup的invalidateChildInParent，如果已经达到最顶层view,则调用ViewRootImpl
            //的invalidateChildInParent。
            parent = parent.invalidateChildInParent(location, dirty);

            if (view != null) {
                // Account for transform on current parent
                Matrix m = view.getMatrix();
                if (!m.isIdentity()) {
                    RectF boundingRect = attachInfo.mTmpTransformRect;
                    boundingRect.set(dirty);
                    m.mapRect(boundingRect);
                    dirty.set((int) (boundingRect.left - 0.5f),
                            (int) (boundingRect.top - 0.5f),
                            (int) (boundingRect.right + 0.5f),
                            (int) (boundingRect.bottom + 0.5f));
                }
            }
        } while (parent != null);
    }
}
```

可以看到，在该方法内部，先设置当前视图的标记位，接着有一个do…while…循环，该循环的作用主要是不断向上回溯父容器，求得父容器和子View需重绘的区域的并集(dirty)。当父容器不是ViewRootImpl的时候，调用的是ViewGroup的invalidateChildInParent方法，我们来看看这个方法，**ViewGroup#invalidateChildInParent:**

```
public ViewParent invalidateChildInParent(final int[] location, final Rect dirty) {
    if ((mPrivateFlags & PFLAG_DRAWN) == PFLAG_DRAWN ||
            (mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == PFLAG_DRAWING_CACHE_VALID) {
        if ((mGroupFlags & (FLAG_OPTIMIZE_INVALIDATE | FLAG_ANIMATION_DONE)) !=
                FLAG_OPTIMIZE_INVALIDATE) {

            //将dirty中的坐标转化为父容器中的坐标，考虑mScrollX和mScrollY的影响
            dirty.offset(location[CHILD_LEFT_INDEX] - mScrollX,
                    location[CHILD_TOP_INDEX] - mScrollY);

            if ((mGroupFlags & FLAG_CLIP_CHILDREN) == 0) {
                //求并集，结果是把子视图的dirty区域转化为父容器的dirty区域
                dirty.union(0, 0, mRight - mLeft, mBottom - mTop);
            }

            final int left = mLeft;
            final int top = mTop;

            if ((mGroupFlags & FLAG_CLIP_CHILDREN) == FLAG_CLIP_CHILDREN) {
                if (!dirty.intersect(0, 0, mRight - left, mBottom - top)) {
                    dirty.setEmpty();
                }
            }
            mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;

            //记录当前视图的mLeft和mTop值，在下一次循环中会把当前值再向父容器的坐标转化
            location[CHILD_LEFT_INDEX] = left;
            location[CHILD_TOP_INDEX] = top;

            if (mLayerType != LAYER_TYPE_NONE) {
                mPrivateFlags |= PFLAG_INVALIDATED;
            }
            //返回当前视图的父容器
            return mParent;

        }
        ...
    }
    return null;
}
```

可以看出，这个方法做的工作主要有：调用offset方法，把当前dirty区域的坐标转化为父容器中的坐标，接着调用union方法，把子dirty区域与父容器的区求并集，换句话说，dirty区域变成父容器区域。最后返回当前视图的父容器，以便进行下一次循环。

回到上面所说的do…while…循环，由于不断向上调用父容器的方法，到最后会调用到ViewRootImpl的invalidateChildInParent方法，我们来看看它的源码，**ViewRootImpl#invalidateChildInParent**:

```
@Override
public ViewParent invalidateChildInParent(int[] location, Rect dirty) {
    checkThread();
    if (DEBUG_DRAW) Log.v(TAG, "Invalidate child: " + dirty);

    if (dirty == null) {
        invalidate();
        return null;
    } else if (dirty.isEmpty() && !mIsAnimating) {
        return null;
    }

    if (mCurScrollY != 0 || mTranslator != null) {
        mTempRect.set(dirty);
        dirty = mTempRect;
        if (mCurScrollY != 0) {
            dirty.offset(0, -mCurScrollY);
        }
        if (mTranslator != null) {
            mTranslator.translateRectInAppWindowToScreen(dirty);
        }
        if (mAttachInfo.mScalingRequired) {
            dirty.inset(-1, -1);
        }
    }

    final Rect localDirty = mDirty;
    if (!localDirty.isEmpty() && !localDirty.contains(dirty)) {
        mAttachInfo.mSetIgnoreDirtyState = true;
        mAttachInfo.mIgnoreDirtyState = true;
    }

    // Add the new dirty rect to the current one
    localDirty.union(dirty.left, dirty.top, dirty.right, dirty.bottom);
    // Intersect with the bounds of the window to skip
    // updates that lie outside of the visible region
    final float appScale = mAttachInfo.mApplicationScale;
    final boolean intersected = localDirty.intersect(0, 0,
            (int) (mWidth * appScale + 0.5f), (int) (mHeight * appScale + 0.5f));
    if (!intersected) {
        localDirty.setEmpty();
    }
    if (!mWillDrawSoon && (intersected || mIsAnimating)) {
        scheduleTraversals();
    }
    return null;
}
```

可以看出，该方法所做的工作与上面的差不多，都进行了offset和union对坐标的调整，然后把dirty区域的信息保存在mDirty中，最后调用了scheduleTraversals方法，触发View的工作流程，由于没有添加measure和layout的标记位，因此measure、layout流程不会执行，而是直接从draw流程开始。

好了，现在总结一下invalidate方法，当子View调用了invalidate方法后，会为该View添加一个标记位，同时不断向父容器请求刷新，父容器通过计算得出身需要重绘的区域，直到传递到ViewRootImpl中，最终触发performTraversals方法，进行开始View树重绘流程(只绘制需要重绘的视图)。

postInvalidate

这个方法与invalidate方法的作用是一样的，都是使View树重绘，但两者的使用条件不同，postInvalidate是在非UI线程中调用，invalidate则是在UI线程中调用。

接下来我们分析postInvalidate方法的原理。

首先看**View#postInvalidate**：

```
public void postInvalidate() {
    postInvalidateDelayed(0);
}

public void postInvalidateDelayed(long delayMilliseconds) {
    // We try only with the AttachInfo because there's no point in invalidating
    // if we are not attached to our window
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        attachInfo.mViewRootImpl.dispatchInvalidateDelayed(this, delayMilliseconds);
    }
}
```

由以上代码可以看出，只有attachInfo不为null的时候才会继续执行，即只有确保视图被添加到窗口的时候才会通知view树重绘，因为这是一个异步方法，果在视图还未被添加到窗口就通知重绘的话会出现错误，所以这样要做一下判断。接着调用了**ViewRootImpl#dispatchInvalidateDelayed**方法：

```
public void dispatchInvalidateDelayed(View view, long delayMilliseconds) {
    Message msg = mHandler.obtainMessage(MSG_INVALIDATE, view);
    mHandler.sendMessageDelayed(msg, delayMilliseconds);
}
```

这里用了Handler，发送了一个异步消息到主线程，显然这里发送的是MSG_INVALIDATE，即通知主线程刷新视图，具体的实现逻辑我们可以看看该

mHandler的实现：

```
final ViewRootHandler mHandler = new ViewRootHandler();

final class ViewRootHandler extends Handler {
        @Override
        public String getMessageName(Message message) {
            ....
        }

        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
            case MSG_INVALIDATE:
                ((View) msg.obj).invalidate();
                break;
            ...
        }
    }
}
```

可以看出，参数message传递过来的正是View视图的实例，然后直接调用了invalidate方法，然后继续invalidate流程。

到目前为止，对于常用的刷新视图的方法已经分析完毕。最后以一幅流程图来说明requestLayout、invalidate的区别：

一般来说，如果View确定自身不再适合当前区域，比如说它的LayoutParams发生了改变，需要父布局对其进行重新测量、布局、绘制这三个流程，往往使用requestLayout。而invalidate则是刷新当前View，使当前View进行重绘，不会进行测量、布局流程，因此如果View只需要重绘而不需要测量，布局的时候，使用invalidate方法往往比requestLayout方法更高效。最后，感谢你们的阅读，希望这篇文章给你们带来帮助。

> 更多阅读
> Android View 测量流程(Measure)完全解析
> Android View 布局流程(Layout)完全解析
> Android View 绘制流程(Draw) 完全解析

- 上一篇 Android View 绘制流程(Draw) 完全解析

我的同类文章

Android（27）

- Android View 绘制流程(Draw) 完全解析 2016-06-02 阅读 348
- Android View 测量流程(Measure)完全解析 2016-05-24 阅读 5174
- 教你轻松实现Material Design风格的知… 2016-05-17 阅读 4819
- 揭开RecyclerView的神秘面纱(二)：处理… 2016-05-11 阅读 1349
- Android ToolBar 使用完全解析 2016-05-07 阅读 1263
- Android View 布局流程(Layout)完全解析 2016-05-28 阅读 319
- Android View源码解读：浅谈DecorView… 2016-05-22 阅读 520
- 揭开RecyclerView的神秘面纱(三)：操作… 2016-05-12 阅读 219
- 揭开RecyclerView的神秘面纱(一)：Rec… 2016-05-09 阅读 1098
- Android View 事件分发机制源码详解(Vi… 2016-05-02 阅读 3832

更多文章

参考知识库

Android知识库

14499 关注 | 1530 收录

猜你在找

- Android入门实战教程
- HTML 5移动开发从入门到精通
- Android高手进阶
- iOS移动开发从入门到精通(Xcode…
- 韦东山嵌入式Linux第一期视频
- android view requestLayout invalid…
- Android view中的requestLayout和i…
- Android view中的requestLayout和i…
- Android view中的requestLayout和i…
- Android中View的requestLayout与i…

查看评论

长城Great 1楼 2016-06-05 01:06发表

牛

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap