android.provider.Settings 自定义安卓系统 Provider

deepwaterooo

May 16, 2022

Contents

1 Android.provider.Settings ContentProvider 又到了弄不懂的地方了

1.1 非常实用的/应用层已经用到的 stub 实现

```
/**
 * Convenience function for updating a single settings value as an
 * integer. This will either create a new entry in the table if the
 * given name does not exist, or modify the value of the existing row
 * with that name.  Note that internally setting values are always
 * stored as strings, so this function converts the given value to a
 * string before storing it.
 *
 * @param cr The ContentResolver to access.
 * @param name The name of the setting to modify.
 * @param value The new value for the setting.
 * @return true if the value was set, false on database errors
 */
public static boolean putInt(ContentResolver cr, String name, int value) { // 添加或是更新一条数据
    return putIntForUser(cr, name, value, cr.getUserId());
}
/** @hide */
@UnsupportedAppUsage
public static boolean putIntForUser(ContentResolver cr, String name, int value,
                                    int userHandle) { // 这里就加上了用户相关的信息，系统自动分用户处理
    return putStringForUser(cr, name, Integer.toString(value), userHandle);
}
/** @hide */
@UnsupportedAppUsage
public static boolean putStringForUser(ContentResolver resolver, String name, String value,
                                       int userHandle) {
    return putStringForUser(resolver, name, value, userHandle,
                            DEFAULT_OVERRIDEABLE_BY_RESTORE);
}
// 下面的方法：没有看懂到底是怎么回事：什么时候转到 Secure，什么时候转到 Global ??? 什么时候可以改，什么时候不能改 ???
private static boolean putStringForUser(ContentResolver resolver, String name, String value,
                                        int userHandle, boolean overrideableByRestore) {
    if (MOVED_TO_SECURE.contains(name)) {
        Log.w(TAG, "Setting " + name + " has moved from android.provider.Settings.System"
```

```
                + " to android.provider.Settings.Secure, value is unchanged.");
            return false;
        }
        if (MOVED_TO_GLOBAL.contains(name) || MOVED_TO_SECURE_THEN_GLOBAL.contains(name)) {
            Log.w(TAG, "Setting " + name + " has moved from android.provider.Settings.System"
                    + " to android.provider.Settings.Global, value is unchanged.");
            return false;
        }
        return sNameValueCache.putStringForUser(resolver, name, value, null, false, userHandle,
                                                overrideableByRestore);
    }
```

1.2 Settings.java 中相对比较重要的类: 1000 lines, tmply paste here

```
@UnsupportedAppUsage(maxTargetSdk = Build.VERSION_CODES.R, trackingBug = 170729553)
private static final ContentProviderHolder sProviderHolder = // 为什么每个类需要自定义自己的 ContentProviderHolder?
    new ContentProviderHolder(CONTENT_URI);

// Populated lazily, guarded by class object:
@UnsupportedAppUsage // 这个东西是需要看懂的
private static final NameValueCache sNameValueCache = new NameValueCache(
    CONTENT_URI,
    CALL_METHOD_GET_GLOBAL,
    CALL_METHOD_PUT_GLOBAL,
    sProviderHolder,
    Global.class);

// Certain settings have been moved from global to the per-user secure namespace
@UnsupportedAppUsage
private static final HashSet<String> MOVED_TO_SECURE;
static {
    MOVED_TO_SECURE = new HashSet<>(8);
    MOVED_TO_SECURE.add(Global.INSTALL_NON_MARKET_APPS);
    MOVED_TO_SECURE.add(Global.ZEN_DURATION);
    MOVED_TO_SECURE.add(Global.SHOW_ZEN_UPGRADE_NOTIFICATION);
    MOVED_TO_SECURE.add(Global.SHOW_ZEN_SETTINGS_SUGGESTION);
    MOVED_TO_SECURE.add(Global.ZEN_SETTINGS_UPDATED);
    MOVED_TO_SECURE.add(Global.ZEN_SETTINGS_SUGGESTION_VIEWED);
    MOVED_TO_SECURE.add(Global.CHARGING_SOUNDS_ENABLED);
    MOVED_TO_SECURE.add(Global.CHARGING_VIBRATION_ENABLED);
    MOVED_TO_SECURE.add(Global.NOTIFICATION_BUBBLES);
}

/** @hide */
public static void getMovedToSecureSettings(Set<String> outKeySet) {
    outKeySet.addAll(MOVED_TO_SECURE);
}

/** @hide */
public static void clearProviderForTest() {
    sProviderHolder.clearProviderForTest();
    sNameValueCache.clearGenerationTrackerForTest();
}

/** @hide */
public static void getPublicSettings(Set<String> allKeys, Set<String> readableKeys,
                                     ArrayMap<String, Integer> readableKeysWithMaxTargetSdk) {
    getPublicSettingsForClass(Global.class, allKeys, readableKeys,
                              readableKeysWithMaxTargetSdk);
}

 // Look up a name in the database.
 // @param resolver to access the database with
 // @param name to look up in the table
 // @return the corresponding value, or null if not present
public static String getString(ContentResolver resolver, String name) {
    return getStringForUser(resolver, name, resolver.getUserId());
}

/** @hide */
@UnsupportedAppUsage(maxTargetSdk = Build.VERSION_CODES.R, trackingBug = 170729553)
public static String getStringForUser(ContentResolver resolver, String name,
                                      int userHandle) {
    if (MOVED_TO_SECURE.contains(name)) {
```

```
        Log.w(TAG, "Setting " + name + " has moved from android.provider.Settings.Global"
              + " to android.provider.Settings.Secure, returning read-only value.");
        return Secure.getStringForUser(resolver, name, userHandle);
    }
    return sNameValueCache.getStringForUser(resolver, name, userHandle);
}

 // Store a name/value pair into the database.
 // @param resolver to access the database with
 // @param name to store
 // @param value to associate with the name
 // @return true if the value was set, false on database errors
public static boolean putString(ContentResolver resolver,
                                String name, String value) {
    return putStringForUser(resolver, name, value, null, false, resolver.getUserId(),
                            DEFAULT_OVERRIDEABLE_BY_RESTORE);
}

 // Store a name/value pair into the database.
 // @param resolver to access the database with
 // @param name to store
 // @param value to associate with the name
 // @param tag to associated with the setting.
 // @param makeDefault whether to make the value the default one.
 // @param overrideableByRestore whether restore can override this value
 // @return true if the value was set, false on database errors
 // @hide@RequiresPermission(Manifest.permission.MODIFY_SETTINGS_OVERRIDEABLE_BY_RESTORE)
public static boolean putString(@NonNull ContentResolver resolver,
                                @NonNull String name, @Nullable String value, @Nullable String tag,
                                boolean makeDefault, boolean overrideableByRestore) {
    return putStringForUser(resolver, name, value, tag, makeDefault,
                            resolver.getUserId(), overrideableByRestore);
}

 // Store a name/value pair into the database.
 // <p>
 // The method takes an optional tag to associate with the setting
 // which can be used to clear only settings made by your package and
 // associated with this tag by passing the tag to {@link
 // #resetToDefaults(ContentResolver, String)}. Anyone can override
 // the current tag. Also if another package changes the setting
 // then the tag will be set to the one specified in the set call
 // which can be null. Also any of the settings setters that do not
 // take a tag as an argument effectively clears the tag.
 // </p><p>
 // For example, if you set settings A and B with tags T1 and T2 and
 // another app changes setting A (potentially to the same value), it
 // can assign to it a tag T3 (note that now the package that changed
 // the setting is not yours). Now if you reset your changes for T1 and
 // T2 only setting B will be reset and A not (as it was changed by
 // another package) but since A did not change you are in the desired
 // initial state. Now if the other app changes the value of A (assuming
 // you registered an observer in the beginning) you would detect that
 // the setting was changed by another app and handle this appropriately
 // (ignore, set back to some value, etc).
 // </p><p>
 // Also the method takes an argument whether to make the value the
 // default for this setting. If the system already specified a default
 // value, then the one passed in here will <strong>not</strong>
 // be set as the default.
 // </p>
 // @param resolver to access the database with.
 // @param name to store.
 // @param value to associate with the name.
 // @param tag to associated with the setting.
 // @param makeDefault whether to make the value the default one.
 // @return true if the value was set, false on database errors.
 // @see #resetToDefaults(ContentResolver, String)
 // @hide@SystemApi
@RequiresPermission(Manifest.permission.WRITE_SECURE_SETTINGS)
public static boolean putString(@NonNull ContentResolver resolver,
                                @NonNull String name, @Nullable String value, @Nullable String tag,
                                boolean makeDefault) {
    return putStringForUser(resolver, name, value, tag, makeDefault,
                            resolver.getUserId(), DEFAULT_OVERRIDEABLE_BY_RESTORE);
```

```
}

 // Reset the settings to their defaults. This would reset <strong>only</strong>
 // settings set by the caller's package. Think of it of a way to undo your own
 // changes to the secure settings. Passing in the optional tag will reset only
 // settings changed by your package and associated with this tag.
 // @param resolver Handle to the content resolver.
 // @param tag Optional tag which should be associated with the settings to reset.
 // @see #putString(ContentResolver, String, String, String, boolean)
 // @hide@SystemApi
@RequiresPermission(Manifest.permission.WRITE_SECURE_SETTINGS)
public static void resetToDefaults(@NonNull ContentResolver resolver,
                                   @Nullable String tag) {
    resetToDefaultsAsUser(resolver, tag, RESET_MODE_PACKAGE_DEFAULTS,
                          resolver.getUserId());
}

 // Reset the settings to their defaults for a given user with a specific mode. The
 // optional tag argument is valid only for {@link #RESET_MODE_PACKAGE_DEFAULTS}
 // allowing resetting the settings made by a package and associated with the tag.
 // @param resolver Handle to the content resolver.
 // @param tag Optional tag which should be associated with the settings to reset.
 // @param mode The reset mode.
 // @param userHandle The user for which to reset to defaults.
 // @see #RESET_MODE_PACKAGE_DEFAULTS
 // @see #RESET_MODE_UNTRUSTED_DEFAULTS
 // @see #RESET_MODE_UNTRUSTED_CHANGES
 // @see #RESET_MODE_TRUSTED_DEFAULTS
 // @hide
public static void resetToDefaultsAsUser(@NonNull ContentResolver resolver,
                                         @Nullable String tag, @ResetMode int mode, @IntRange(from = 0) int userHandle) {
    try {
        Bundle arg = new Bundle();
        arg.putInt(CALL_METHOD_USER_KEY, userHandle);
        if (tag != null) {
            arg.putString(CALL_METHOD_TAG_KEY, tag);
        }
        arg.putInt(CALL_METHOD_RESET_MODE_KEY, mode);
        IContentProvider cp = sProviderHolder.getProvider(resolver);
        cp.call(resolver.getAttributionSource(),
                sProviderHolder.mUri.getAuthority(), CALL_METHOD_RESET_GLOBAL, null, arg);
    } catch (RemoteException e) {
        Log.w(TAG, "Can't reset do defaults for " + CONTENT_URI, e);
    }
}

/** @hide */
@UnsupportedAppUsage(maxTargetSdk = Build.VERSION_CODES.R, trackingBug = 170729553)
public static boolean putStringForUser(ContentResolver resolver,
                                       String name, String value, int userHandle) {
    return putStringForUser(resolver, name, value, null, false, userHandle,
                            DEFAULT_OVERRIDEABLE_BY_RESTORE);
}

/** @hide */
public static boolean putStringForUser(@NonNull ContentResolver resolver,
                                       @NonNull String name, @Nullable String value, @Nullable String tag,
                                       boolean makeDefault, @UserIdInt int userHandle, boolean overrideableByRestore) {
    if (LOCAL_LOGV) {
        Log.v(TAG, "Global.putString(name=" + name + ", value=" + value
              + " for " + userHandle);
    }
    // Global and Secure have the same access policy so we can forward writes
    if (MOVED_TO_SECURE.contains(name)) {
        Log.w(TAG, "Setting " + name + " has moved from android.provider.Settings.Global"
              + " to android.provider.Settings.Secure, value is unchanged.");
        return Secure.putStringForUser(resolver, name, value, tag,
                                       makeDefault, userHandle, overrideableByRestore);
    }
    return sNameValueCache.putStringForUser(resolver, name, value, tag,
                                            makeDefault, userHandle, overrideableByRestore);
}

 // Construct the content URI for a particular name/value pair,
 // useful for monitoring changes with a ContentObserver.
```

```
 // @param name to look up in the table
 // @return the corresponding content URI, or null if not present
public static Uri getUriFor(String name) {
    return getUriFor(CONTENT_URI, name);
}

 // Convenience function for retrieving a single secure settings value
 // as an integer.  Note that internally setting values are always
 // stored as strings; this function converts the string to an integer
 // for you.  The default value will be returned if the setting is
 // not defined or not an integer.
 // @param cr The ContentResolver to access.
 // @param name The name of the setting to retrieve.
 // @param def Value to return if the setting is not defined.
 // @return The setting's current value, or 'def' if it is not defined
 // or not a valid integer.public static int getInt(ContentResolver cr, String name, int def) {
    String v = getString(cr, name);
    try {
        return v != null ? Integer.parseInt(v) : def;
    } catch (NumberFormatException e) {
        return def;
    }
}

 // Convenience function for retrieving a single secure settings value
 // as an integer.  Note that internally setting values are always
 // stored as strings; this function converts the string to an integer
 // for you.
 // <p>
 // This version does not take a default value.  If the setting has not
 // been set, or the string value is not a number,
 // it throws {@link SettingNotFoundException}.
 // @param cr The ContentResolver to access.
 // @param name The name of the setting to retrieve.
 // @throws SettingNotFoundException Thrown if a setting by the given
 // name can't be found or the setting value is not an integer.
 // @return The setting's current value.
public static int getInt(ContentResolver cr, String name)
    throws SettingNotFoundException {
    String v = getString(cr, name);
    try {
        return Integer.parseInt(v);
    } catch (NumberFormatException e) {
        throw new SettingNotFoundException(name);
    }
}

 // Convenience function for updating a single settings value as an
 // integer. This will either create a new entry in the table if the
 // given name does not exist, or modify the value of the existing row
 // with that name.  Note that internally setting values are always
 // stored as strings, so this function converts the given value to a
 // string before storing it.
 // @param cr The ContentResolver to access.
 // @param name The name of the setting to modify.
 // @param value The new value for the setting.
 // @return true if the value was set, false on database errors
public static boolean putInt(ContentResolver cr, String name, int value) {
    return putString(cr, name, Integer.toString(value));
}

 // Convenience function for retrieving a single secure settings value
 // as a {@code long}.  Note that internally setting values are always
 // stored as strings; this function converts the string to a {@code long}
 // for you.  The default value will be returned if the setting is
 // not defined or not a {@code long}.
 // @param cr The ContentResolver to access.
 // @param name The name of the setting to retrieve.
 // @param def Value to return if the setting is not defined.
 // @return The setting's current value, or 'def' if it is not defined
 // or not a valid {@code long}.
public static long getLong(ContentResolver cr, String name, long def) {
    String valString = getString(cr, name);
    long value;
    try {
```

```
        value = valString != null ? Long.parseLong(valString) : def;
    } catch (NumberFormatException e) {
        value = def;
    }
    return value;
}

 // Convenience function for retrieving a single secure settings value
 // as a {@code long}.  Note that internally setting values are always
 // stored as strings; this function converts the string to a {@code long}
 // for you.
 // <p>
 // This version does not take a default value.  If the setting has not
 // been set, or the string value is not a number,
 // it throws {@link SettingNotFoundException}.
 // @param cr The ContentResolver to access.
 // @param name The name of the setting to retrieve.
 // @return The setting's current value.
 // @throws SettingNotFoundException Thrown if a setting by the given
 // name can't be found or the setting value is not an integer.
public static long getLong(ContentResolver cr, String name)
    throws SettingNotFoundException {
    String valString = getString(cr, name);
    try {
        return Long.parseLong(valString);
    } catch (NumberFormatException e) {
        throw new SettingNotFoundException(name);
    }
}

 // Convenience function for updating a secure settings value as a long
 // integer. This will either create a new entry in the table if the
 // given name does not exist, or modify the value of the existing row
 // with that name.  Note that internally setting values are always
 // stored as strings, so this function converts the given value to a
 // string before storing it.
 // @param cr The ContentResolver to access.
 // @param name The name of the setting to modify.
 // @param value The new value for the setting.
 // @return true if the value was set, false on database errors
public static boolean putLong(ContentResolver cr, String name, long value) {
    return putString(cr, name, Long.toString(value));
}

 // Convenience function for retrieving a single secure settings value
 // as a floating point number.  Note that internally setting values are
 // always stored as strings; this function converts the string to an
 // float for you. The default value will be returned if the setting
 // is not defined or not a valid float.
 // @param cr The ContentResolver to access.
 // @param name The name of the setting to retrieve.
 // @param def Value to return if the setting is not defined.
 // @return The setting's current value, or 'def' if it is not defined
 // or not a valid float.
public static float getFloat(ContentResolver cr, String name, float def) {
    String v = getString(cr, name);
    try {
        return v != null ? Float.parseFloat(v) : def;
    } catch (NumberFormatException e) {
        return def;
    }
}

 // Convenience function for retrieving a single secure settings value
 // as a float.  Note that internally setting values are always
 // stored as strings; this function converts the string to a float
 // for you.
 // <p>
 // This version does not take a default value.  If the setting has not
 // been set, or the string value is not a number,
 // it throws {@link SettingNotFoundException}.
 // @param cr The ContentResolver to access.
 // @param name The name of the setting to retrieve.
 // @throws SettingNotFoundException Thrown if a setting by the given
 // name can't be found or the setting value is not a float.
```

```
     // @return The setting's current value.
    public static float getFloat(ContentResolver cr, String name)
        throws SettingNotFoundException {
        String v = getString(cr, name);
        if (v == null) {
            throw new SettingNotFoundException(name);
        }
        try {
            return Float.parseFloat(v);
        } catch (NumberFormatException e) {
            throw new SettingNotFoundException(name);
        }
    }

     // Convenience function for updating a single settings value as a
     // floating point number. This will either create a new entry in the
     // table if the given name does not exist, or modify the value of the
     // existing row with that name.  Note that internally setting values
     // are always stored as strings, so this function converts the given
     // value to a string before storing it.
     // @param cr The ContentResolver to access.
     // @param name The name of the setting to modify.
     // @param value The new value for the setting.
     // @return true if the value was set, false on database errors
    public static boolean putFloat(ContentResolver cr, String name, float value) {
        return putString(cr, name, Float.toString(value));
    }

     // Subscription Id to be used for voice call on a multi sim device.
     // @hide@Readable
    public static final String MULTI_SIM_VOICE_CALL_SUBSCRIPTION = "multi_sim_voice_call";

     // Used to provide option to user to select subscription during dial.
     // The supported values are 0 = disable or 1 = enable prompt.
     // @hide@UnsupportedAppUsage
    @Readable
    public static final String MULTI_SIM_VOICE_PROMPT = "multi_sim_voice_prompt";

     // Subscription Id to be used for data call on a multi sim device.
     // @hide@Readable
    public static final String MULTI_SIM_DATA_CALL_SUBSCRIPTION = "multi_sim_data_call";

     // Subscription Id to be used for SMS on a multi sim device.
     // @hide@Readable
    public static final String MULTI_SIM_SMS_SUBSCRIPTION = "multi_sim_sms";

     // Used to provide option to user to select subscription during send SMS.
     // The value 1 - enable, 0 - disable
     // @hide@Readable
    public static final String MULTI_SIM_SMS_PROMPT = "multi_sim_sms_prompt";

    // User preferred subscriptions setting.
     // This holds the details of the user selected subscription from the card and
     // the activation status. Each settings string have the comma separated values
     // iccId,appType,appId,activationStatus,3gppIndex,3gpp2Index
     // @hide@UnsupportedAppUsage
    @Readable
    public static final String[] MULTI_SIM_USER_PREFERRED_SUBS = {"user_preferred_sub1",
        "user_preferred_sub2","user_preferred_sub3"};

     // Which subscription is enabled for a physical slot.
     // @hide@Readable
    public static final String ENABLED_SUBSCRIPTION_FOR_SLOT = "enabled_subscription_for_slot";

     // Whether corresponding logical modem is enabled for a physical slot.
     // The value 1 - enable, 0 - disable
     // @hide@Readable
    public static final String MODEM_STACK_ENABLED_FOR_SLOT = "modem_stack_enabled_for_slot";

     // Whether to enable new contacts aggregator or not.
     // The value 1 - enable, 0 - disable
     // @hide@Readable
    public static final String NEW_CONTACT_AGGREGATOR = "new_contact_aggregator";

     // Whether to enable contacts metadata syncing or not
```

```
 // The value 1 - enable, 0 - disable
 // @removed@Deprecated
@Readable
public static final String CONTACT_METADATA_SYNC = "contact_metadata_sync";

 // Whether to enable contacts metadata syncing or not
 // The value 1 - enable, 0 - disable@Readable
public static final String CONTACT_METADATA_SYNC_ENABLED = "contact_metadata_sync_enabled";

 // Whether to enable cellular on boot.
 // The value 1 - enable, 0 - disable
 // @hide@Readable
public static final String ENABLE_CELLULAR_ON_BOOT = "enable_cellular_on_boot";

 // The maximum allowed notification enqueue rate in Hertz.
 // Should be a float, and includes updates only.
 // @hide@Readable
public static final String MAX_NOTIFICATION_ENQUEUE_RATE = "max_notification_enqueue_rate";

 // Displays toasts when an app posts a notification that does not specify a valid channel.
 // The value 1 - enable, 0 - disable
 // @hide@Readable
public static final String SHOW_NOTIFICATION_CHANNEL_WARNINGS =
    "show_notification_channel_warnings";

 // Whether cell is enabled/disabled
 // @hide@Readable
public static final String CELL_ON = "cell_on";

 // Global settings which can be accessed by instant apps.
 // @hidepublic static final Set<String> INSTANT_APP_SETTINGS = new ArraySet<>();
static {
    INSTANT_APP_SETTINGS.add(WAIT_FOR_DEBUGGER);
    INSTANT_APP_SETTINGS.add(DEVICE_PROVISIONED);
    INSTANT_APP_SETTINGS.add(DEVELOPMENT_FORCE_RESIZABLE_ACTIVITIES);
    INSTANT_APP_SETTINGS.add(DEVELOPMENT_FORCE_RTL);
    INSTANT_APP_SETTINGS.add(EPHEMERAL_COOKIE_MAX_SIZE_BYTES);
    INSTANT_APP_SETTINGS.add(AIRPLANE_MODE_ON);
    INSTANT_APP_SETTINGS.add(WINDOW_ANIMATION_SCALE);
    INSTANT_APP_SETTINGS.add(TRANSITION_ANIMATION_SCALE);
    INSTANT_APP_SETTINGS.add(ANIMATOR_DURATION_SCALE);
    INSTANT_APP_SETTINGS.add(DEBUG_VIEW_ATTRIBUTES);
    INSTANT_APP_SETTINGS.add(DEBUG_VIEW_ATTRIBUTES_APPLICATION_PACKAGE);
    INSTANT_APP_SETTINGS.add(WTF_IS_FATAL);
    INSTANT_APP_SETTINGS.add(SEND_ACTION_APP_ERROR);
    INSTANT_APP_SETTINGS.add(ZEN_MODE);
}

 // Whether to show the high temperature warning notification.
 // @hide@Readable
public static final String SHOW_TEMPERATURE_WARNING = "show_temperature_warning";

 // Whether to show the usb high temperature alarm notification.
 // @hide@Readable
public static final String SHOW_USB_TEMPERATURE_ALARM = "show_usb_temperature_alarm";

 // Temperature at which the high temperature warning notification should be shown.
 // @hide@Readable
public static final String WARNING_TEMPERATURE = "warning_temperature";

 // Whether the diskstats logging task is enabled/disabled.
 // @hide@Readable
public static final String ENABLE_DISKSTATS_LOGGING = "enable_diskstats_logging";

 // Whether the cache quota calculation task is enabled/disabled.
 // @hide@Readable
public static final String ENABLE_CACHE_QUOTA_CALCULATION =
    "enable_cache_quota_calculation";

 // Whether the Deletion Helper no threshold toggle is available.
 // @hide@Readable
public static final String ENABLE_DELETION_HELPER_NO_THRESHOLD_TOGGLE =
    "enable_deletion_helper_no_threshold_toggle";

 // The list of snooze options for notifications
```

```
 // This is encoded as a key=value list, separated by commas. Ex:
 // "default=60,options_array=15:30:60:120"
 // The following keys are supported:
 // <pre>
 // default               (int)
 // options_array         (int[])
 // </pre>
 // All delays in integer minutes. Array order is respected.
 // Options will be used in order up to the maximum allowed by the UI.
 // @hide@Readable
public static final String NOTIFICATION_SNOOZE_OPTIONS =
    "notification_snooze_options";

 // When enabled, notifications the notification assistant service has modified will show an
 // indicator. When tapped, this indicator will describe the adjustment made and solicit
 // feedback. This flag will also add a "automatic" option to the long press menu.
 // The value 1 - enable, 0 - disable
 // @hidepublic static final String NOTIFICATION_FEEDBACK_ENABLED = "notification_feedback_enabled";

 // Settings key for the ratio of notification dismissals to notification views - one of the
 // criteria for showing the notification blocking helper.
 // <p>The value is a float ranging from 0.0 to 1.0 (the closer to 0.0, the more intrusive
 // the blocking helper will be).
 // @hide@Readable
public static final String BLOCKING_HELPER_DISMISS_TO_VIEW_RATIO_LIMIT =
    "blocking_helper_dismiss_to_view_ratio";

 // Settings key for the longest streak of dismissals  - one of the criteria for showing the
 // notification blocking helper.
 // <p>The value is an integer greater than 0.
 // @hide@Readable
public static final String BLOCKING_HELPER_STREAK_LIMIT = "blocking_helper_streak_limit";

 // Configuration flags for SQLite Compatibility WAL. Encoded as a key-value list, separated
 // by commas. E.g.: compatibility_wal_supported=true, wal_syncmode=OFF
 // Supported keys:<br/>
 // <li>
 // <ul> {@code legacy_compatibility_wal_enabled} : A {code boolean} flag that determines
 // whether or not "compatibility WAL" mode is enabled by default. This is a legacy flag
 // and is honoured on Android Q and higher. This flag will be removed in a future release.
 // </ul>
 // <ul> {@code wal_syncmode} : A {@code String} representing the synchronization mode to use
 // when WAL is enabled, either via {@code legacy_compatibility_wal_enabled} or using the
 // obsolete {@code compatibility_wal_supported} flag.
 // </ul>
 // <ul> {@code truncate_size} : A {@code int} flag that specifies the truncate size of the
 // WAL journal.
 // </ul>
 // <ul> {@code compatibility_wal_supported} : A {code boolean} flag that specifies whether
 // the legacy "compatibility WAL" mode is enabled by default. This flag is obsolete and is
 // only supported on Android Pie.
 // </ul>
 // </li>
 // @hide@Readable
public static final String SQLITE_COMPATIBILITY_WAL_FLAGS =
    "sqlite_compatibility_wal_flags";

 // Enable GNSS Raw Measurements Full Tracking?
 // 0 = no
 // 1 = yes
 // @hide@Readable
public static final String ENABLE_GNSS_RAW_MEAS_FULL_TRACKING =
    "enable_gnss_raw_meas_full_tracking";

 // Whether the notification should be ongoing (persistent) when a carrier app install is
 // required.
 // The value is a boolean (1 or 0).
 // @hide@SystemApi
@Readable
public static final String INSTALL_CARRIER_APP_NOTIFICATION_PERSISTENT =
    "install_carrier_app_notification_persistent";

 // The amount of time (ms) to hide the install carrier app notification after the user has
 // ignored it. After this time passes, the notification will be shown again
 // The value is a long
```

```
 // @hide@SystemApi
@Readable
public static final String INSTALL_CARRIER_APP_NOTIFICATION_SLEEP_MILLIS =
    "install_carrier_app_notification_sleep_millis";

 // Whether we've enabled zram on this device. Takes effect on
 // reboot. The value "1" enables zram; "0" disables it, and
 // everything else is unspecified.
 // @hide@Readable
public static final String ZRAM_ENABLED =
    "zram_enabled";

 // Whether the app freezer is enabled on this device.
 // The value of "enabled" enables the app freezer, "disabled" disables it and
 // "device_default" will let the system decide whether to enable the freezer or not
 // @hide@Readable
public static final String CACHED_APPS_FREEZER_ENABLED = "cached_apps_freezer";

 // Configuration flags for smart replies in notifications.
 // This is encoded as a key=value list, separated by commas. Ex:
 // "enabled=1,max_squeeze_remeasure_count=3"
 // The following keys are supported:
 // <pre>
 // enabled                          (boolean)
 // requires_targeting_p             (boolean)
 // max_squeeze_remeasure_attempts   (int)
 // edit_choices_before_sending      (boolean)
 // show_in_heads_up                 (boolean)
 // min_num_system_generated_replies (int)
 // max_num_actions                  (int)
 // </pre>
 // @see com.android.systemui.statusbar.policy.SmartReplyConstants
 // @hide@Readable
public static final String SMART_REPLIES_IN_NOTIFICATIONS_FLAGS =
    "smart_replies_in_notifications_flags";

 // Configuration flags for the automatic generation of smart replies and smart actions in
 // notifications. This is encoded as a key=value list, separated by commas. Ex:
 // "generate_replies=false,generate_actions=true".
 // The following keys are supported:
 // <pre>
 // generate_replies                (boolean)
 // generate_actions                (boolean)
 // </pre>
 // @hide@Readable
public static final String SMART_SUGGESTIONS_IN_NOTIFICATIONS_FLAGS =
    "smart_suggestions_in_notifications_flags";

 // If nonzero, crashes in foreground processes will bring up a dialog.
 // Otherwise, the process will be silently killed.
 // @hide@TestApi
@Readable
@SuppressLint("NoSettingsProvider")
public static final String SHOW_FIRST_CRASH_DIALOG = "show_first_crash_dialog";

 // If nonzero, crash dialogs will show an option to restart the app.
 // @hide@Readable
public static final String SHOW_RESTART_IN_CRASH_DIALOG = "show_restart_in_crash_dialog";

 // If nonzero, crash dialogs will show an option to mute all future crash dialogs for
 // this app.
 // @hide@Readable
public static final String SHOW_MUTE_IN_CRASH_DIALOG = "show_mute_in_crash_dialog";


 // If nonzero, will show the zen upgrade notification when the user toggles DND on/off.
 // @hide
 // @deprecated - Use {@link android.provider.Settings.Secure#SHOW_ZEN_UPGRADE_NOTIFICATION}@Deprecated
public static final String SHOW_ZEN_UPGRADE_NOTIFICATION = "show_zen_upgrade_notification";

 // If nonzero, will show the zen update settings suggestion.
 // @hide
 // @deprecated - Use {@link android.provider.Settings.Secure#SHOW_ZEN_SETTINGS_SUGGESTION}@Deprecated
public static final String SHOW_ZEN_SETTINGS_SUGGESTION = "show_zen_settings_suggestion";
```

```
 // If nonzero, zen has not been updated to reflect new changes.
 // @deprecated - Use {@link android.provider.Settings.Secure#ZEN_SETTINGS_UPDATED}
 // @hide@Deprecated
public static final String ZEN_SETTINGS_UPDATED = "zen_settings_updated";

 // If nonzero, zen setting suggestion has been viewed by user
 // @hide
 // @deprecated - Use {@link android.provider.Settings.Secure#ZEN_SETTINGS_SUGGESTION_VIEWED}@Deprecated
public static final String ZEN_SETTINGS_SUGGESTION_VIEWED =
    "zen_settings_suggestion_viewed";

 // Backup and restore agent timeout parameters.
 // These parameters are represented by a comma-delimited key-value list.
 // The following strings are supported as keys:
 // <pre>
 //     kv_backup_agent_timeout_millis         (long)
 //     full_backup_agent_timeout_millis       (long)
 //     shared_backup_agent_timeout_millis     (long)
 //     restore_agent_timeout_millis           (long)
 //     restore_agent_finished_timeout_millis  (long)
 // </pre>
 // They map to milliseconds represented as longs.
 // Ex: "kv_backup_agent_timeout_millis=30000,full_backup_agent_timeout_millis=300000"
 // @hide@Readable
public static final String BACKUP_AGENT_TIMEOUT_PARAMETERS =
    "backup_agent_timeout_parameters";

 // Blocklist of GNSS satellites.
 // This is a list of integers separated by commas to represent pairs of (constellation,
 // svid). Thus, the number of integers should be even.
 // E.g.: "3,0,5,24" denotes (constellation=3, svid=0) and (constellation=5, svid=24) are
 // blocklisted. Note that svid=0 denotes all svids in the constellation are blocklisted.
 // @hidepublic static final String GNSS_SATELLITE_BLOCKLIST = "gnss_satellite_blocklist";

 // Duration of updates in millisecond for GNSS location request from HAL to framework.
 // If zero, the GNSS location request feature is disabled.
 // The value is a non-negative long.
 // @hide@Readable
public static final String GNSS_HAL_LOCATION_REQUEST_DURATION_MILLIS =
    "gnss_hal_location_request_duration_millis";

 // Binder call stats settings.
 // The following strings are supported as keys:
 // <pre>
 //     enabled              (boolean)
 //     detailed_tracking    (boolean)
 //     upload_data          (boolean)
 //     sampling_interval    (int)
 // </pre>
 // @hide@Readable
public static final String BINDER_CALLS_STATS = "binder_calls_stats";

 // Looper stats settings.
 // The following strings are supported as keys:
 // <pre>
 //     enabled              (boolean)
 //     sampling_interval    (int)
 // </pre>
 // @hide@Readable
public static final String LOOPER_STATS = "looper_stats";

 // Settings for collecting statistics on CPU usage per thread
 // The following strings are supported as keys:
 // <pre>
 //     num_buckets          (int)
 //     collected_uids       (string)
 //     minimum_total_cpu_usage_millis (int)
 // </pre>
 // @hide@Readable
public static final String KERNEL_CPU_THREAD_READER = "kernel_cpu_thread_reader";

 // Whether we've enabled native flags health check on this device. Takes effect on
 // reboot. The value "1" enables native flags health check; otherwise it's disabled.
 // @hide@Readable
public static final String NATIVE_FLAGS_HEALTH_CHECK_ENABLED =
```

```
    "native_flags_health_check_enabled";

 // Parameter for {@link #APPOP_HISTORY_PARAMETERS} that controls the mode
 // in which the historical registry operates.
 // @hide@Readable
public static final String APPOP_HISTORY_MODE = "mode";

 // Parameter for {@link #APPOP_HISTORY_PARAMETERS} that controls how long
 // is the interval between snapshots in the base case i.e. the most recent
 // part of the history.
 // @hide@Readable
public static final String APPOP_HISTORY_BASE_INTERVAL_MILLIS = "baseIntervalMillis";

 // Parameter for {@link #APPOP_HISTORY_PARAMETERS} that controls the base
 // for the logarithmic step when building app op history.
 // @hide@Readable
public static final String APPOP_HISTORY_INTERVAL_MULTIPLIER = "intervalMultiplier";

 // Appop history parameters. These parameters are represented by
 // a comma-delimited key-value list.
 // The following strings are supported as keys:
 // <pre>
 //     mode                  (int)
 //     baseIntervalMillis    (long)
 //     intervalMultiplier    (int)
 // </pre>
 // Ex: "mode=HISTORICAL_MODE_ENABLED_ACTIVE,baseIntervalMillis=1000,intervalMultiplier=10"
 // @see #APPOP_HISTORY_MODE
 // @see #APPOP_HISTORY_BASE_INTERVAL_MILLIS
 // @see #APPOP_HISTORY_INTERVAL_MULTIPLIER
 // @hide@Readable
public static final String APPOP_HISTORY_PARAMETERS =
    "appop_history_parameters";

 // Auto revoke parameters. These parameters are represented by
 // a comma-delimited key-value list.
 // <pre>
 //     enabledForPreRApps    (bolean)
 //     unusedThresholdMs     (long)
 //     checkFrequencyMs      (long)
 // </pre>
 // Ex: "enabledForPreRApps=false,unusedThresholdMs=7776000000,checkFrequencyMs=1296000000"
 // @hide@Readable
public static final String AUTO_REVOKE_PARAMETERS =
    "auto_revoke_parameters";

 // Delay for sending ACTION_CHARGING after device is plugged in.
 // This is used as an override for constants defined in BatteryStatsImpl for
 // ease of experimentation.
 // @see com.android.internal.os.BatteryStatsImpl.Constants.KEY_BATTERY_CHARGED_DELAY_MS
 // @hide@Readable
public static final String BATTERY_CHARGING_STATE_UPDATE_DELAY =
    "battery_charging_state_update_delay";

 // A serialized string of params that will be loaded into a text classifier action model.
 // @hide@Readable
public static final String TEXT_CLASSIFIER_ACTION_MODEL_PARAMS =
    "text_classifier_action_model_params";

 // The amount of time to suppress "power-off" from the power button after the device has
 // woken due to a gesture (lifting the phone).  Since users have learned to hit the power
 // button immediately when lifting their device, it can cause the device to turn off if a
 // gesture has just woken the device. This value tells us the milliseconds to wait after
 // a gesture before "power-off" via power-button is functional again. A value of 0 is no
 // delay, and reverts to the old behavior.
 // @hide@Readable
public static final String POWER_BUTTON_SUPPRESSION_DELAY_AFTER_GESTURE_WAKE =
    "power_button_suppression_delay_after_gesture_wake";

 // The usage amount of advanced battery. The value is 0~100.
 // @hide@Readable
public static final String ADVANCED_BATTERY_USAGE_AMOUNT = "advanced_battery_usage_amount";

 // For 5G NSA capable devices, determines whether NR tracking indications are on
 // when the screen is off.
```

```
 // Values are:
 // 0: off - All 5G NSA tracking indications are off when the screen is off.
 // 1: extended - All 5G NSA tracking indications are on when the screen is off as long as
 //    the device is camped on 5G NSA (5G icon is showing in status bar).
 //    If the device is not camped on 5G NSA, tracking indications are off.
 // 2: always on - All 5G NSA tracking indications are on whether the screen is on or off.
 // @hide@Readable
public static final String NR_NSA_TRACKING_SCREEN_OFF_MODE =
    "nr_nsa_tracking_screen_off_mode";

 // Whether to show People Space.
 // Values are:
 // 0: Disabled (default)
 // 1: Enabled
 // @hide
public static final String SHOW_PEOPLE_SPACE = "show_people_space";

 // Which types of conversation(s) to show in People Space.
 // Values are:
 // 0: Single user-selected conversation (default)
 // 1: Priority conversations only
 // 2: All conversations
 // @hide
public static final String PEOPLE_SPACE_CONVERSATION_TYPE =
    "people_space_conversation_type";

 // Whether to show new notification dismissal.
 // Values are:
 // 0: Disabled
 // 1: Enabled
 // @hide
public static final String SHOW_NEW_NOTIF_DISMISS = "show_new_notif_dismiss";

 // Block untrusted touches mode.
 // Can be one of:
 // <ul>
 //      <li>0 = {@link BlockUntrustedTouchesMode#DISABLED}: Feature is off.
 //      <li>1 = {@link BlockUntrustedTouchesMode#PERMISSIVE}: Untrusted touches are flagged
 //          but not blocked
 //      <li>2 = {@link BlockUntrustedTouchesMode#BLOCK}: Untrusted touches are blocked
 // </ul>
 // @hide@Readable
public static final String BLOCK_UNTRUSTED_TOUCHES_MODE = "block_untrusted_touches";

 // The maximum allowed obscuring opacity by UID to propagate touches.
 // For certain window types (eg. SAWs), the decision of honoring {@link LayoutParams
 // #FLAG_NOT_TOUCHABLE} or not depends on the combined obscuring opacity of the windows
 // above the touch-consuming window.
 // For a certain UID:
 // <ul>
 //     <li>If it's the same as the UID of the touch-consuming window, allow it to propagate
 //     the touch.
 //     <li>Otherwise take all its windows of eligible window types above the touch-consuming
 //     window, compute their combined obscuring opacity considering that {@code
 //     opacity(A, B) = 1 - (1 - opacity(A))*(1 - opacity(B))}. If the computed value is
 //     lesser than or equal to this setting and there are no other windows preventing the
 //     touch, allow the UID to propagate the touch.
 // </ul>
 // @see android.hardware.input.InputManager#getMaximumObscuringOpacityForTouch()
 // @see android.hardware.input.InputManager#setMaximumObscuringOpacityForTouch(float)
 // @hide@Readable
public static final String MAXIMUM_OBSCURING_OPACITY_FOR_TOUCH =
    "maximum_obscuring_opacity_for_touch";

 // Used to enable / disable the Restricted Networking Mode in which network access is
 // restricted to apps holding the CONNECTIVITY_USE_RESTRICTED_NETWORKS permission.
 // Values are:
 // 0: disabled
 // 1: enabled
 // @hide
public static final String RESTRICTED_NETWORKING_MODE = "restricted_networking_mode";
}

// Whether or not syncs (bulk set operations) for {@link DeviceConfig} are disabled
// currently. The value is boolean (1 or 0). The value '1' means that {@link
```

```
// DeviceConfig#setProperties(DeviceConfig.Properties)} will return {@code false}.
// @hide
public static final String DEVICE_CONFIG_SYNC_DISABLED = "device_config_sync_disabled";

/** @hide */ public static String zenModeToString(int mode) {
    if (mode == ZEN_MODE_IMPORTANT_INTERRUPTIONS) return "ZEN_MODE_IMPORTANT_INTERRUPTIONS";
    if (mode == ZEN_MODE_ALARMS) return "ZEN_MODE_ALARMS";
    if (mode == ZEN_MODE_NO_INTERRUPTIONS) return "ZEN_MODE_NO_INTERRUPTIONS";
    return "ZEN_MODE_OFF";
}
/** @hide */ public static boolean isValidZenMode(int value) {
    switch (value) {
    case Global.ZEN_MODE_OFF:
    case Global.ZEN_MODE_IMPORTANT_INTERRUPTIONS:
    case Global.ZEN_MODE_ALARMS:
    case Global.ZEN_MODE_NO_INTERRUPTIONS:
        return true;
    default:
        return false;
    }
}
```

```
@Mock
BluetoothAdapter bluetoothAdapter;
@Mock
BluetoothLeScanner bluetoothLeScanner;
@Mock
PackageManager packageManager;
@Mock
Context context;

@Before
public void setUp() throws Exception {
    ShadowLog.stream = System.out;
}

private void setUpMocksForAdapter() {
    BtScanner.testBtleSupport = true;
    AndroidBtManager.setBluetoothAdpater(bluetoothAdapter);
    when(bluetoothAdapter.isEnabled()).thenReturn(true);
    when(bluetoothAdapter.getState()).thenReturn(BluetoothAdapter.STATE_ON);
    when(bluetoothAdapter.getBluetoothLeScanner()).thenReturn(bluetoothLeScanner);
    when(context.getPackageManager()).thenReturn(packageManager);
    when(packageManager.hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)).thenReturn(true);
}

private long startTime = 0;
private long time = 0;
private ScannerCountDownLatch countDownLatch;
private int btDiscoveryTime;
private int btleScanTime;

@Test
public void testBtScannerCycle_BtleScanlength() throws InterruptedException
{
    StatusEventCallback statusEventCallback = new StatusEventCallback()
    {
        @Override
        public void onStatusEvent(StatusEvent statusEvent, PhdInformation phdInformation)
        {
            System.out.println("Status event " + statusEvent.name());
            switch(statusEvent)
            {
            case BTLE_SCAN_STARTED:
                startTime = System.currentTimeMillis();
                break;
            case CLASSIC_SCAN_STARTED:
                time = System.currentTimeMillis() - startTime - btleScanTime;
                startTime = System.currentTimeMillis();
                countDownLatch.countDown();
                break;
            }
        }
```

```
    };
    btDiscoveryTime = 1000;
    btleScanTime = 10000;
    Context context = ApplicationProvider.getApplicationContext();
    setUpMocksForAdapter();
    AndroidBtManager.setStatusEventCallback(statusEventCallback);
    BtScanner.setScanTimes(btDiscoveryTime, btleScanTime);
    BtScanner btScanner = new BtScanner(context, bluetoothAdapter);
    startTime  = System.currentTimeMillis();
    countDownLatch = new ScannerCountDownLatch(2);
    btScanner.start();
    shadowOf(Looper.getMainLooper()).idle();
    if(countDownLatch.await(15, TimeUnit.SECONDS)) // Assert length of btle scan time 1
    {
        assertTrue((time > -50 && time < 50));
        countDownLatch = new ScannerCountDownLatch(1);
        if(countDownLatch.await(15, TimeUnit.SECONDS))  // Assert length of btle scan time 2
        {
            assertTrue((time > -50 && time < 50));
        }
        else // Timeout
        {
            fail();
        }
    }
    else  // Timeout
    {
        fail();
    }
    btScanner.terminateScan();
    assertFalse(btScanner.isBtScannerRunning());
}
```

1.3 BaseColumns.java interface

```
package android.provider;
public interface BaseColumns {
    /**
     * The unique ID for a row.
     * <p>Type: INTEGER (long)</p>
     */
    // @Column(Cursor.FIELD_TYPE_INTEGER)
    public static final String _ID = "_id";
    /**
     * The count of rows in a directory.
     *
     * <p>Type: INTEGER</p>
     */
    // @Column(Cursor.FIELD_TYPE_INTEGER)
    public static final String _COUNT = "_count";
}
```

1.4 Settings.NameValueTable class: ContentProvider 中的键值对表格定义

```
    // package android.provider;
    // public final class Settings {
    /**
     * Common base for tables of name/value settings.
     */
    public static class NameValueTable implements BaseColumns {
        public static final String NAME = "name";
        public static final String VALUE = "value";
        // A flag indicating whether the current value of a setting should be preserved during restore.
        /** @hide */
        public static final String IS_PRESERVED_IN_RESTORE = "is_preserved_in_restore";

        protected static boolean putString(ContentResolver resolver, Uri uri, String name, String value) {
            // The database will take care of replacing duplicates.
            try {
                ContentValues values = new ContentValues();
```

```
            values.put(NAME, name);
            values.put(VALUE, value);
            resolver.insert(uri, values);
            return true;
        } catch (SQLException e) {
            Log.w(TAG, "Can't set key " + name + " in " + uri, e);
            return false;
        }
    }
    public static Uri getUriFor(Uri uri, String name) {
        return Uri.withAppendedPath(uri, name);
    }
}
```

1.5 Settings.NameValueTable 类的多个不同的继承者：System，Secure，Global，Config，我们在应用中使用到的是 Global

```
// public static final class System extends NameValueTable {
// public static final class Secure extends NameValueTable {
// public static final class Global extends NameValueTable {
// public static final class Config extends NameValueTable {
```

- 部分起头源码

```
/**
 * System settings, containing miscellaneous system preferences.  This
 * table holds simple name/value pairs.  There are convenience
 * functions for accessing individual settings entries.
 */
public static final class System extends NameValueTable {
    // NOTE: If you add new settings here, be sure to add them to
    // com.android.providers.settings.SettingsProtoDumpUtil#dumpProtoSystemSettingsLocked.
    private static final float DEFAULT_FONT_SCALE = 1.0f;
    private static final int DEFAULT_FONT_WEIGHT = 0;

    /**
     * The content:// style URL for this table
     */
    public static final Uri CONTENT_URI =
        Uri.parse("content://" + AUTHORITY + "/system");
    // .......
}
/**
 * Secure system settings, containing system preferences that applications
 * can read but are not allowed to write.  These are for preferences that
 * the user must explicitly modify through the UI of a system app. Normal
 * applications cannot modify the secure settings database, either directly
 * or by calling the "put" methods that this class contains.
 */
public static final class Secure extends NameValueTable {
    // NOTE: If you add new settings here, be sure to add them to
    // com.android.providers.settings.SettingsProtoDumpUtil#dumpProtoSecureSettingsLocked.
    /**
     * The content:// style URL for this table
     */
    public static final Uri CONTENT_URI =
        Uri.parse("content://" + AUTHORITY + "/secure");
    @UnsupportedAppUsage(maxTargetSdk = Build.VERSION_CODES.R, trackingBug = 170729553)
        private static final ContentProviderHolder sProviderHolder =
        new ContentProviderHolder(CONTENT_URI);
    // Populated lazily, guarded by class object:
    @UnsupportedAppUsage
        private static final NameValueCache sNameValueCache = new NameValueCache(
            CONTENT_URI,
            CALL_METHOD_GET_SECURE,
            CALL_METHOD_PUT_SECURE,
            sProviderHolder,
            Secure.class);
    private static ILockSettings sLockSettings = null;
    // .......
}
public static final String AUTHORITY = "settings";
```

```
    /**
     * Global system settings, containing preferences that always apply identically
     * to all defined users.  Applications can read these but are not allowed to write;
     * like the "Secure" settings, these are for preferences that the user must
     * explicitly modify through the system UI or specialized APIs for those values.
     */
    public static final class Global extends NameValueTable {
        // NOTE: If you add new settings here, be sure to add them to
        // com.android.providers.settings.SettingsProtoDumpUtil#dumpProtoGlobalSettingsLocked.
        /**
         * The content:// style URL for global secure settings items.  Not public.
         */
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/global"); // content://settings/glob
        // .......
    }
    /**
     * Configuration system settings, containing settings which are applied identically for all
     * defined users. Only Android can read these and only a specific configuration service can
     * write these.
     *
     * @hide
     */
    public static final class Config extends NameValueTable {
        /**
         * The modes that can be used when disabling syncs to the 'config' settings.
         * @hide
         */
        @IntDef(prefix = "DISABLE_SYNC_MODE_",
                value = { SYNC_DISABLED_MODE_NONE, SYNC_DISABLED_MODE_PERSISTENT,
                    SYNC_DISABLED_MODE_UNTIL_REBOOT })
            @Retention(RetentionPolicy.SOURCE)
            @Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
            public @interface SyncDisabledMode {}
        /**
         * Sync is not not disabled.
         *
         * @hide
         */
        public static final int SYNC_DISABLED_MODE_NONE = 0;
        /**
         * Disabling of Config bulk update / syncing is persistent, i.e. it survives a device
         * reboot.
         * @hide
         */
        public static final int SYNC_DISABLED_MODE_PERSISTENT = 1;
        /**
         * Disabling of Config bulk update / syncing is not persistent, i.e. it will not survive a
         * device reboot.
         * @hide
         */
        public static final int SYNC_DISABLED_MODE_UNTIL_REBOOT = 2;
        private static final ContentProviderHolder sProviderHolder =
            new ContentProviderHolder(DeviceConfig.CONTENT_URI);
        // .......
    }
```

1.6 Settings.java

```
package android.provider;
public final class Settings {
/** @hide */
public static final boolean DEFAULT_OVERRIDEABLE_BY_RESTORE = false;

private static final class ContentProviderHolder {
    private final Object mLock = new Object();
    private final Uri mUri;
    @GuardedBy("mLock")
    @UnsupportedAppUsage
    private IContentProvider mContentProvider;
    public ContentProviderHolder(Uri uri) {
        mUri = uri;
    }
    public IContentProvider getProvider(ContentResolver contentResolver) {
        synchronized (mLock) {
```

```
            if (mContentProvider == null) {
                mContentProvider = contentResolver.acquireProvider(mUri.getAuthority());
            }
            return mContentProvider;
        }
    }
    public void clearProviderForTest() {
        synchronized (mLock) {
            mContentProvider = null;
        }
    }
}

// Thread-safe.
private static class NameValueCache { // 500 lines of code, too much .......
    private static final boolean DEBUG = false;

    private static final String[] SELECT_VALUE_PROJECTION = new String[] {
        Settings.NameValueTable.VALUE
    };

    private static final String NAME_EQ_PLACEHOLDER = "name=?";

    // Must synchronize on 'this' to access mValues and mValuesVersion.
    private final ArrayMap<String, String> mValues = new ArrayMap<>();

    private final Uri mUri;
    @UnsupportedAppUsage
    private final ContentProviderHolder mProviderHolder;

    // The method we'll call (or null, to not use) on the provider
    // for the fast path of retrieving settings.
    private final String mCallGetCommand;
    private final String mCallSetCommand;
    private final String mCallListCommand;
    private final String mCallSetAllCommand;

    private final ArraySet<String> mReadableFields;
    private final ArraySet<String> mAllFields;
    private final ArrayMap<String, Integer> mReadableFieldsWithMaxTargetSdk;

    @GuardedBy("this")
    private GenerationTracker mGenerationTracker;

    <T extends NameValueTable> NameValueCache(Uri uri, String getCommand,
                                              String setCommand, ContentProviderHolder providerHolder, Class<T> callerClass
        this(uri, getCommand, setCommand, null, null, providerHolder, callerClass);
    }
    private <T extends NameValueTable> NameValueCache(Uri uri, String getCommand,
                                                      String setCommand, String listCommand, String setAllCommand,
                                                      ContentProviderHolder providerHolder, Class<T> callerClass) {
        mUri = uri;
        mCallGetCommand = getCommand;
        mCallSetCommand = setCommand;
        mCallListCommand = listCommand;
        mCallSetAllCommand = setAllCommand;
        mProviderHolder = providerHolder;
        mReadableFields = new ArraySet<>();
        mAllFields = new ArraySet<>();
        mReadableFieldsWithMaxTargetSdk = new ArrayMap<>();
        getPublicSettingsForClass(callerClass, mAllFields, mReadableFields,
                                  mReadableFieldsWithMaxTargetSdk);
    }
    // .....
}
```