Copy and Paste

Quickview

- A clipboard-based framework for copying and pasting data.
- Supports both simple and complex data, including text strings, complex data structures, text and binary stream data, and application assets.
- Copies and pastes simple text directly to and from the clipboard.
- Copies and pastes complex data using a content provider.
- Requires API 11.

In this document

Key classes

- `ClipboardManager`
- `ClipData`
- `ClipData.Item`
- `ClipDescription`
- `Uri`
- `ContentProvider`
- `Intent`

Related Samples

- Note Pad sample application

See also

- Content Providers

Android provides a powerful clipboard-based framework for copying and pasting. It supports both simple and complex data types, including text strings, complex data structures, text and binary stream data, and even application assets. Simple text data is stored directly in the clipboard, while complex data is stored as a reference that the pasting application resolves with a content provider. Copying and pasting works both within an application and between applications that implement the framework.

Since a part of the framework uses content providers, this topic assumes some familiarity with the Android Content Provider API, which is described in the topic Content Providers.

The Clipboard Framework

When you use the clipboard framework, you put data into a clip object, and then put the clip object on the system-wide clipboard. The clip object can take one of three forms:

Text

> A text string. You put the string directly into the clip object, which you then put onto the clipboard. To paste the string, you get the clip object from the clipboard and copy the string to into your application's storage.

URI

> A `Uri` object representing any form of URI. This is primarily for copying complex data from a content provider. To copy data, you put a `Uri` object into a clip object and put the clip object onto the clipboard. To paste the data, you get the clip object, get the `Uri` object, resolve it to a data source such as a content provider, and copy the data from the source into your application's storage.

Intent

> An `Intent`. This supports copying application shortcuts. To copy data, you create an Intent, put it into a clip object, and put the clip object onto the clipboard. To paste the data, you get the clip object and then copy the Intent object into your application's memory area.

The clipboard holds only one clip object at a time. When an application puts a clip object on the clipboard, the previous clip object disappears.

If you want to allow users to paste data into your application, you don't have to handle all types of data. You can examine the data on the clipboard before you give users the option to paste it. Besides having a certain data form, the clip object also contains metadata that tells you what MIME type or types are available. This metadata helps you decide if your application can do something useful with the clipboard data. For example, if you have an application that primarily handles text, you may want to ignore clip objects that contain a URI or Intent.

You may also want to allow users to paste text regardless of the form of data on the clipboard. To do this, you can force the clipboard data into a text representation, and then paste this text. This is described in the section Coercing the clipboard to text.

Clipboard Classes

This section describes the classes used by the clipboard framework.

ClipboardManager

In the Android system, the system clipboard is represented by the global `ClipboardManager` class. You do not instantiate this class directly; instead, you get a reference to it by invoking `getSystemService(CLIPBOARD_SERVICE)`.

ClipData, ClipData.Item, and ClipDescription

To add data to the clipboard, you create a `ClipData` object that contains both a description of the data and the data itself. The clipboard holds only one `ClipData` at a time. A `ClipData` contains a `ClipDescription` object and one or more `ClipData.Item` objects.

A `ClipDescription` object contains metadata about the clip. In particular, it contains an array of available MIME types for the clip's data. When you put a clip on the clipboard, this array is available to pasting applications, which can examine it to see if they can handle any of available the MIME types.

A `ClipData.Item` object contains the text, URI, or Intent data:

Text

A `CharSequence`.

URI

A `Uri`. This usually contains a content provider URI, although any URI is allowed. The application that provides the data puts the URI on the clipboard. Applications that want to paste the data get the URI from the clipboard and use it to access the content provider (or other data source) and retrieve the data.

Intent

An `Intent`. This data type allows you to copy an application shortcut to the clipboard. Users can then paste the shortcut into their applications for later use.

You can add more than one `ClipData.Item` object to a clip. This allows users to copy and paste multiple selections as a single clip. For example, if you have a list widget that allows the user to select more than one item at a time, you can copy all the items to the clipboard at once. To do this, you create a separate `ClipData.Item` for each list item, and then you add the `ClipData.Item` objects to the `ClipData` object.

ClipData convenience methods

The `ClipData` class provides static convenience methods for creating a `ClipData` object with a single `ClipData.Item` object and a simple `ClipDescription` object:

`newPlainText(label, text)`

Returns a `ClipData` object whose single `ClipData.Item` object contains a text string. The `ClipDescription` object's label is set to `label`. The single MIME type in `ClipDescription` is `MIMETYPE_TEXT_PLAIN`.

Use `newPlainText()` to create a clip from a text string.

`newUri(resolver, label, URI)`

Returns a `ClipData` object whose single `ClipData.Item` object contains a URI. The `ClipDescription` object's label is set to `label`. If the URI is a content URI (`Uri.getScheme()` returns `content:`), the method uses the `ContentResolver` object provided in `resolver` to retrieve the available MIME types from the content provider and store them in `ClipDescription`. For a URI that is not a `content:` URI, the method sets the MIME type to `MIMETYPE_TEXT_URILIST`.

Use `newUri()` to create a clip from a URI, particularly a `content:` URI.

`newIntent(label, intent)`

Returns a `ClipData` object whose single `ClipData.Item` object contains an `Intent`. The `ClipDescription` object's label is set to `label`. The MIME type is set to `MIMETYPE_TEXT_INTENT`.

Use `newIntent()` to create a clip from an Intent object.

Coercing the clipboard data to text

Even if your application only handles text, you can copy non-text data from the clipboard by converting it with the method `ClipData.Item.coerceToText()`.

This method converts the data in `ClipData.Item` to text and returns a `CharSequence`. The value that `ClipData.Item.coerceToText()` returns is based on the form of data in `ClipData.Item`:

Text

If `ClipData.Item` is text (`getText()` is not null), `coerceToText()` returns the text.

URI

If `ClipData.Item` is a URI (`getUri()` is not null), `coerceToText()` tries to use it as a content URI:

- If the URI is a content URI and the provider can return a text stream, `coerceToText()` returns a text stream.
- If the URI is a content URI but the provider does not offer a text stream, `coerceToText()` returns a representation of the URI. The representation is the same as that returned by `Uri.toString()`.
- If the URI is not a content URI, `coerceToText()` returns a representation of the URI. The representation is the same as that returned by `Uri.toString()`.

Intent

If `ClipData.Item` is an Intent (`getIntent()` is not null), `coerceToText()` converts it to an Intent URI and returns it. The representation is the same as that returned by `Intent.toUri(URI_INTENT_SCHEME)`.

The clipboard framework is summarized in Figure 1. To copy data, an application puts a `ClipData` object on the `ClipboardManager` global clipboard. The `ClipData` contains one or more `ClipData.Item` objects and one `ClipDescription` object. To paste data, an application gets the `ClipData`, gets its MIME type from the `ClipDescription`, and gets the data either from the `ClipData.Item` or from the content provider referred to by `ClipData.Item`.

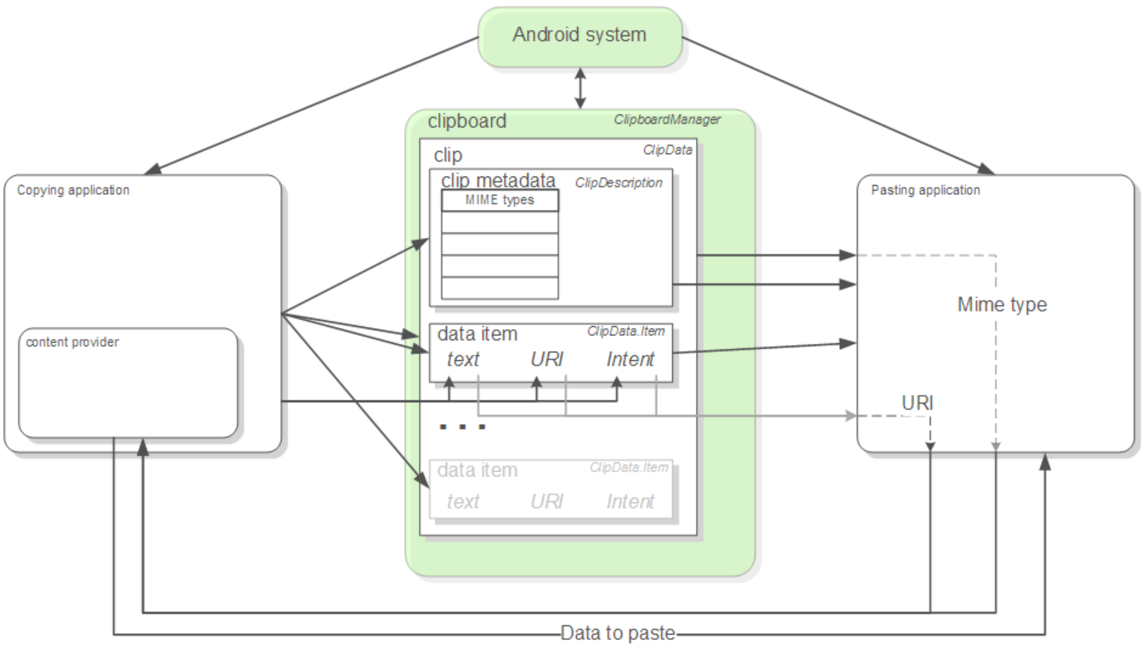

Figure 1. The Android clipboard framework

Copying to the Clipboard

As described previously, to copy data to the clipboard you get a handle to the global `ClipboardManager` object, create a `ClipData` object, add a `ClipDescription` and one or more `ClipData.Item` objects to it, and add the finished `ClipData` object to the `ClipboardManager` object. This is described in detail in the following procedure:

1. If you are copying data using a content URI, set up a content provider.

   The Note Pad sample application is an example of using a content provider for copying and pasting. The NotePadProvider class implements the content provider. The NotePad class defines a contract between the provider and other applications, including the supported MIME types.

2. Get the system clipboard:

```
...

// if the user selects copy
case R.id.menu_copy:

// Gets a handle to the clipboard service.
ClipboardManager clipboard = (ClipboardManager)
        getSystemService(Context.CLIPBOARD_SERVICE);
```

3. Copy the data to a new `ClipData` object:
    - For text

      ```
      // Creates a new text clip to put on the clipboard
      ClipData clip = ClipData.newPlainText("simple text", "Hello, World!");
      ```

    - For a URI

      This snippet constructs a URI by encoding a record ID onto the content URI for the provider. This technique is covered in more detail in the section Encoding an identifier on the URI:

      ```
      // Creates a Uri based on a base Uri and a record ID based on the contact's last name
      // Declares the base URI string
      private static final String CONTACTS = "content://com.example.contacts";

      // Declares a path string for URIs that you use to copy data
      private static final String COPY_PATH = "/copy";

      // Declares the Uri to paste to the clipboard
      Uri copyUri = Uri.parse(CONTACTS + COPY_PATH + "/" + lastName);

      ...

      // Creates a new URI clip object. The system uses the anonymous getContentResolver() object to
      // get MIME types from provider. The clip object's label is "URI", and its data is
      // the Uri previously created.
      ClipData clip = ClipData.newUri(getContentResolver(), "URI", copyUri);
      ```

    - For an Intent

      This snippet constructs an Intent for an application and then puts it in the clip object:

      ```
      // Creates the Intent
      Intent appIntent = new Intent(this, com.example.demo.myapplication.class);

      ...

      // Creates a clip object with the Intent in it. Its label is "Intent" and its data is
      // the Intent object created previously
      ClipData clip = ClipData.newIntent("Intent", appIntent);
      ```

4. Put the new clip object on the clipboard:

   ```
   // Set the clipboard's primary clip.
   clipboard.setPrimaryClip(clip);
   ```

Pasting from the Clipboard

As described previously, you paste data from the clipboard by getting the global clipboard object, getting the clip object, looking at its data, and if possible copying the data from the clip object to your own storage. This section describes in detail how to do this for the three forms of clipboard data.

Pasting plain text

To paste plain text, first get the global clipboard and verify that it can return plain text. Then get the clip object and copy its text to your own storage using `getText()`, as described in the following procedure:

1. Get the global `ClipboardManager` object using `getSystemService(CLIPBOARD_SERVICE)`. Also declare a global variable to contain the pasted text:

```
ClipboardManager clipboard = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);
String pasteData = "";
```

2. Next, determine if you should enable or disable the "paste" option in the current Activity. You should verify that the clipboard contains a clip and that you can handle the type of data represented by the clip:

```
// Gets the ID of the "paste" menu item
MenuItem mPasteItem = menu.findItem(R.id.menu_paste);

// If the clipboard doesn't contain data, disable the paste menu item.
// If it does contain data, decide if you can handle the data.
if (!(clipboard.hasPrimaryClip())) {

    mPasteItem.setEnabled(false);

    } else if (!(clipboard.getPrimaryClipDescription().hasMimeType(MIMETYPE_TEXT_PLAIN))) {

        // This disables the paste menu item, since the clipboard has data but it is not plain text
        mPasteItem.setEnabled(false);
    } else {

        // This enables the paste menu item, since the clipboard contains plain text.
        mPasteItem.setEnabled(true);
    }
}
```

3. Copy the data from the clipboard. This point in the program is only reachable if the "paste" menu item is enabled, so you can assume that the clipboard contains plain text. You do not yet know if it contains a text string or a URI that points to plain text. The following snippet tests this, but it only shows the code for handling plain text:

```
// Responds to the user selecting "paste"
case R.id.menu_paste:

// Examines the item on the clipboard. If getText() does not return null, the clip item contains the
// text. Assumes that this application can only handle one item at a time.
 ClipData.Item item = clipboard.getPrimaryClip().getItemAt(0);

// Gets the clipboard as text.
pasteData = item.getText();

// If the string contains data, then the paste operation is done
if (pasteData != null) {
    return;

// The clipboard does not contain text. If it contains a URI, attempts to get data from it
} else {
    Uri pasteUri = item.getUri();

    // If the URI contains something, try to get text from it
    if (pasteUri != null) {

        // calls a routine to resolve the URI and get data from it. This routine is not
        // presented here.
        pasteData = resolveUri(Uri);
        return;
    } else {

    // Something is wrong. The MIME type was plain text, but the clipboard does not contain either
    // text or a Uri. Report an error.
    Log.e("Clipboard contains an invalid data type");
    return;
    }
}
```

Pasting data from a content URI

If the ClipData.Item object contains a content URI and you have determined that you can handle one of its MIME types, create a ContentResolver and then call the appropriate content provider method to retrieve the data.

The following procedure describes how to get data from a content provider based on a content URI on the clipboard. It checks that a MIME type that the application can use is available from the provider:

1. Declare a global variable to contain the MIME type:

```
// Declares a MIME type constant to match against the MIME types offered by the provider
public static final String MIME_TYPE_CONTACT = "vnd.android.cursor.item/vnd.example.contact";
```

2. Get the global clipboard. Also get a content resolver so you can access the content provider:

```
// Gets a handle to the Clipboard Manager
ClipboardManager clipboard = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);

// Gets a content resolver instance
ContentResolver cr = getContentResolver();
```

3. Get the primary clip from the clipboard, and get its contents as a URI:

```
// Gets the clipboard data from the clipboard
ClipData clip = clipboard.getPrimaryClip();

if (clip != null) {

    // Gets the first item from the clipboard data
    ClipData.Item item = clip.getItemAt(0);

    // Tries to get the item's contents as a URI
    Uri pasteUri = item.getUri();
```

4. Test to see if the URI is a content URI by calling getType(Uri). This method returns null if Uri does not point to a valid content provider:

```
    // If the clipboard contains a URI reference
    if (pasteUri != null) {

        // Is this a content URI?
        String uriMimeType = cr.getType(pasteUri);
```

5. Test to see if the content provider supports a MIME type that the current application understands. If it does, call ContentResolver.query() to get the data. The return value is a Cursor:

```
        // If the return value is not null, the Uri is a content Uri
        if (uriMimeType != null) {

            // Does the content provider offer a MIME type that the current application can use?
            if (uriMimeType.equals(MIME_TYPE_CONTACT)) {

                // Get the data from the content provider.
                Cursor pasteCursor = cr.query(uri, null, null, null, null);

                // If the Cursor contains data, move to the first record
                if (pasteCursor != null) {
                    if (pasteCursor.moveToFirst()) {

                    // get the data from the Cursor here. The code will vary according to the
                    // format of the data model.
                    }
                }

                // close the Cursor
                pasteCursor.close();
             }
         }
     }
}
```

Pasting an Intent

To paste an Intent, first get the global clipboard. Examine the `ClipData.Item` object to see if it contains an Intent. Then call `getIntent()` to copy the Intent to your own storage. The following snippet demonstrates this:

```
// Gets a handle to the Clipboard Manager
ClipboardManager clipboard = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);

// Checks to see if the clip item contains an Intent, by testing to see if getIntent() returns null
Intent pasteIntent = clipboard.getPrimaryClip().getItemAt(0).getIntent();

if (pasteIntent != null) {

    // handle the Intent

} else {

    // ignore the clipboard, or issue an error if your application was expecting an Intent to be
    // on the clipboard
}
```

Using Content Providers to Copy Complex Data

Content providers support copying complex data such as database records or file streams. To copy the data, you put a content URI on the clipboard. Pasting applications then get this URI from the clipboard and use it to retrieve database data or file stream descriptors.

Since the pasting application only has the content URI for your data, it needs to know which piece of data to retrieve. You can provide this information by encoding an identifier for the data on the URI itself, or you can provide a unique URI that will return the data you want to copy. Which technique you choose depends on the organization of your data.

The following sections describe how to set up URIs, how to provide complex data, and how to provide file streams. The descriptions assume that you are familiar with the general principles of content provider design.

Encoding an identifier on the URI

A useful technique for copying data to the clipboard with a URI is to encode an identifier for the data on the URI itself. Your content provider can then get the identifier from the URI and use it to retrieve the data. The pasting application doesn't have to know that the identifier exists; all it has to do is get your "reference" (the URI plus the identifier) from the clipboard, give it your content provider, and get back the data.

You usually encode an identifier onto a content URI by concatenating it to the end of the URI. For example, suppose you define your provider URI as the following string:

```
"content://com.example.contacts"
```

If you want to encode a name onto this URI, you would use the following snippet:

```
String uriString = "content://com.example.contacts" + "/" + "Smith";

// uriString now contains content://com.example.contacts/Smith.

// Generates a uri object from the string representation
Uri copyUri = Uri.parse(uriString);
```

If you are already using a content provider, you may want to add a new URI path that indicates the URI is for copying. For example, suppose you already have the following URI paths:

```
"content://com.example.contacts"/people
"content://com.example.contacts"/people/detail
"content://com.example.contacts"/people/images
```

You could add another path that is specific to copy URIs:

```
"content://com.example.contacts/copying"
```

You could then detect a "copy" URI by pattern-matching and handle it with code that is specific for copying and pasting.

You normally use the encoding technique if you're already using a content provider, internal database, or internal table to organize your data. In these cases, you have multiple pieces of data you want to copy, and presumably a unique identifier for each piece. In response to a query from the pasting application, you can look up the data by its identifier and return it.

If you don't have multiple pieces of data, then you probably don't need to encode an identifier. You can simply use a URI that is unique to your provider. In response to a query, your provider would return the data it currently contains.

Getting a single record by ID is used in the Note Pad sample application to open a note from the notes list. The sample uses the `_id` field from an SQL database, but you can have any numeric or character identifier you want.

Copying data structures

You set up a content provider for copying and pasting complex data as a subclass of the `ContentProvider` component. You should also encode the URI you put on the clipboard so that it points to the exact record you want to provide. In addition, you have to consider the existing state of your application:

- If you already have a content provider, you can add to its functionality. You may only need to modify its `query()` method to handle URIs coming from applications that want to paste data. You will probably want to modify the method to handle a "copy" URI pattern.
- If your application maintains an internal database, you may want to move this database into a content provider to facilitate copying from it.
- If you are not currently using a database, you can implement a simple content provider whose sole purpose is to offer data to applications that are pasting from the clipboard.

In the content provider, you will want to override at least the following methods:

`query()`

Pasting applications will assume that they can get your data by using this method with the URI you put on the clipboard. To support copying, you should have this method detect URIs that contain a special "copy" path. Your application can then create a "copy" URI to put on the clipboard, containing the copy path and a pointer to the exact record you want to copy.

`getType()`

This method should return the MIME type or types for the data you intend to copy. The method `newUri()` calls `getType()` in order to put the MIME types into the new `ClipData` object.

MIME types for complex data are described in the topic Content Providers.

Notice that you don't have to have any of the other content provider methods such as `insert()` or `update()`. A pasting application only needs to get your supported MIME types and copy data from your provider. If you already have these methods, they won't interfere with copy operations.

The following snippets demonsrate how to set up your application to copy complex data:

1. In the global constants for your application, declare a base URI string and a path that identifies URI strings you are using to copy data. Also declare a MIME type for the copied data:

```
// Declares the base URI string
private static final String CONTACTS = "content://com.example.contacts";

// Declares a path string for URIs that you use to copy data
private static final String COPY_PATH = "/copy";

// Declares a MIME type for the copied data
public static final String MIME_TYPE_CONTACT = "vnd.android.cursor.item/vnd.example.contact";
```

2. In the Activity from which users copy data, set up the code to copy data to the clipboard. In response to a copy request, put the URI on the clipboard:

```
public class MyCopyActivity extends Activity {

    ...

// The user has selected a name and is requesting a copy.
case R.id.menu_copy:

    // Appends the last name to the base URI
    // The name is stored in "lastName"
    uriString = CONTACTS + COPY_PATH + "/" + lastName;

    // Parses the string into a URI
    Uri copyUri = Uri.parse(uriString);

    // Gets a handle to the clipboard service.
    ClipboardManager clipboard = (ClipboardManager)
        getSystemService(Context.CLIPBOARD_SERVICE);

    ClipData clip = ClipData.newUri(getContentResolver(), "URI", copyUri);

    // Set the clipboard's primary clip.
    clipboard.setPrimaryClip(clip);
```

3. In the global scope of your content provider, create a URI matcher and add a URI pattern that will match URIs you put on the clipboard:

```
public class MyCopyProvider extends ContentProvider {

    ...

// A Uri Match object that simplifies matching content URIs to patterns.
private static final UriMatcher sURIMatcher = new UriMatcher(UriMatcher.NO_MATCH);

// An integer to use in switching based on the incoming URI pattern
private static final int GET_SINGLE_CONTACT = 0;

...

// Adds a matcher for the content URI. It matches
// "content://com.example.contacts/copy/*"
sUriMatcher.addURI(CONTACTS, "names/*", GET_SINGLE_CONTACT);
```

4. Set up the `query()` method. This method can handle different URI patterns, depending on how you code it, but only the pattern for the

clipboard copying operation is shown:

```
// Sets up your provider's query() method.
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
    String sortOrder) {

    ...

    // Switch based on the incoming content URI
    switch (sUriMatcher.match(uri)) {

    case GET_SINGLE_CONTACT:

        // query and return the contact for the requested name. Here you would decode
        // the incoming URI, query the data model based on the last name, and return the result
        // as a Cursor.

    ...

}
```

5. Set up the `getType()` method to return an appropriate MIME type for copied data:

```
// Sets up your provider's getType() method.
public String getType(Uri uri) {
    ...

    switch (sUriMatcher.match(uri)) {
    case GET_SINGLE_CONTACT:
        return (MIME_TYPE_CONTACT);
    ...
    }
}
```

The section Pasting data from a content URI describes how to get a content URI from the clipboard and use it to get and paste data.

Copying data streams

You can copy and paste large amounts of text and binary data as streams. The data can have forms such as the following:

- Files stored on the actual device.
- Streams from sockets.
- Large amounts of data stored in a provider's underlying database system.

A content provider for data streams provides access to its data with a file descriptor object such as `AssetFileDescriptor` instead of a `Cursor` object. The pasting application reads the data stream using this file descriptor.

To set up your application to copy a data stream with a provider, follow these steps:

1. Set up a content URI for the data stream you are putting on the clipboard. Options for doing this include the following:
    - Encode an identifier for the data stream onto the URI, as described in the section Encoding an identifier on the URI, and then maintain a table in your provider that contains identifiers and the corresponding stream name.
    - Encode the stream name directly on the URI.
    - Use a unique URI that always returns the current stream from the provider. If you use this option, you have to remember to update your provider to point to a different stream whenever you copy the stream to the clipboard via the URI.
2. Provide a MIME type for each type of data stream you plan to offer. Pasting applications need this information to determine if they can paste the data on the clipboard.
3. Implement one of the `ContentProvider` methods that returns a file descriptor for a stream. If you encode identifiers on the content URI, use this method to determine which stream to open.
4. To copy the data stream to the clipboard, construct the content URI and place it on the clipboard.

To paste a data stream, an application gets the clip from the clipboard, gets the URI, and uses it in a call to a `ContentResolver` file descriptor method that opens the stream. The `ContentResolver` method calls the corresponding `ContentProvider` method, passing it the content URI. Your provider returns the file descriptor to `ContentResolver` method. The pasting application then has the responsibility to read the data from the stream.

The following list shows the most important file descriptor methods for a content provider. Each of these has a corresponding `ContentResolver` method with the string "Descriptor" appended to the method name; for example, the `ContentResolver` analog of `openAssetFile()` is `openAssetFileDescriptor()`:

`openTypedAssetFile()`

> This method should return an asset file descriptor, but only if the provided MIME type is supported by the provider. The caller (the application doing the pasting) provides a MIME type pattern. The content provider (of the application that has copied a URI to the clipboard) returns an `AssetFileDescriptor` file handle if it can provide that MIME type, or throws an exception if it can not.
>
> This method handles subsections of files. You can use it to read assets that the content provider has copied to the clipboard.

`openAssetFile()`

> This method is a more general form of `openTypedAssetFile()`. It does not filter for allowed MIME types, but it can read subsections of files.

`openFile()`

> This is a more general form of `openAssetFile()`. It can't read subsections of files.

You can optionally use the `openPipeHelper()` method with your file descriptor method. This allows the pasting application to read the stream data in a background thread using a pipe. To use this method, you need to implement the `ContentProvider.PipeDataWriter` interface. An example of doing this is given in the Note Pad sample application, in the `openTypedAssetFile()` method of `NotePadProvider.java`.

Designing Effective Copy/Paste Functionality

To design effective copy and paste functionality for your application, remember these points:

- At any time, there is only one clip on the clipboard. A new copy operation by any application in the system overwrites the previous clip. Since the user may navigate away from your application and do a copy before returning, you can't assume that the clipboard contains the clip that the user previously copied in *your* application.
- The intended purpose of multiple `ClipData.Item` objects per clip is to support copying and pasting of multiple selections rather than different forms of reference to a single selection. You usually want all of the `ClipData.Item` objects in a clip to have the same form, that is, they should all be simple text, content URI, or `Intent`, but not a mixture.
- When you provide data, you can offer different MIME representations. Add the MIME types you support to the `ClipDescription`, and then implement the MIME types in your content provider.
- When you get data from the clipboard, your application is responsible for checking the available MIME types and then deciding which one, if any, to use. Even if there is a clip on the clipboard and the user requests a paste, your application is not required to do the paste. You *should* do the paste if the MIME type is compatible. You may choose to coerce the data on the clipboard to text using `coerceToText()` if you choose. If your application supports more than one of the available MIME types, you can allow the user to choose which one to use.