

第六章 记录存储系统

如果用户正在进行激烈的游戏，此时正好有重要的事情，需要退出程序去打电话，这个时候必须要保存当前的进度，下次还能够继续游戏。或者对于得分类的游戏，由于用户需要比较每次游戏的得分，这就需要存储每次游戏的分数。还有需要保存游戏当前的进度，在出现错误操作的时候可以重新再玩。所有这些都需要能够对数据进行存储。

由于 MIDP 中程序是以 jar 的形式打包存储的，所以在 jar 中无法写入数据，因此系统单独开辟了存储空间用来存储数据，以及管理存储空间的管理系统。

6.1 记录存储系统概述

MIDP 中的存储系统实际上是实现为一个类似于数据库的系统，并不是简单的文件系统，称为记录管理系统（Record Management System，RMS）。

6.1.1 记录存储的概念

MIDP 中的 MIDlet 为了实现数据的持久性管理，提供了数据存储的功能，可以在程序下次启动的时候再次使用。这种持久性存储机制，叫做记录管理系统（RMS），是一个简单的面向记录的数据库模型。

记录存储是面向记录的数据库，可把一个记录存储看作一个数据库表文件，有许多记录组合而成，这些记录将持久保存并支持跨多个 MIDlet 的请求。在系统平台的整个常规应用期间，包括重启、更换电池等，MIDlet 的记录存储都由系统平台负责维护，系统会尽可能维持记录的完整性。一般情况下，不会出现记录丢失的情况。

记录存储的生成位置是由平台决定的，并不会暴露给 MIDlet。在 MIDlet 组件范围内可见的记录存储命名是简单命名方式，就是说只依赖于打开或创建存储时给予的名称。一个 MIDlet 可以创建多个名字不同的记录存储。当一个 MIDlet 组件从系统删除的时候，所有与此 MIDlet 组件有关的存储记录应当都被删除，此删除操作不通过用户手工进行，而是系统自动删除。在同一个 MIDlet 组件内的 MIDlet 可以相互之间直接存取。而且 MIDP2.0 对于 MIDP1.0 增加了允许不同 MIDlet 组件之间可以相互访问的存储记录的 API，但是在必须在创建存储记录的时候给与足够的权限。共享能力通过其他 MIDlet 套件的记录名和定义的存取权限来获得。

同一 MIDlet 组件内记录存储的命名必须唯一。不同 MIDlet 组件内的记录存储名称可以相同，这是由于系统内记录存储的命名是使用 MIDlet 组件名加记录名的方式。组件名是通过应用描述中的 MIDlet-Vendor 和 MIDlet-Name 属性来标识的。记录存储的名字是大小写敏感的，可以由最多 32 个 Unicode 字符组成。

记录存储系统并没有提供锁机制来进行记录存储的同步共享机制。但是记录存储的实现确保所有单个记录存储操作是原子的，同步的和序列化的，通过此种方式来保证同时访问的时候不会发生错误。但是如果一个 MIDlet 使用多线程来存取记录，需要由 MIDlet 来保证此次存取的同步，也就是说可能会产生一个未定义顺序的存储结果。比如说，如果两个线程同时并发调用 RecordStore.setRecord 在一个记录存储上，记录存储会序列化这些调用，并不会发生数据库的损坏错误。但是其中的一个写操作可能会被接下来的另一个操作重写，这可能会在 MIDlet 内引起问题。如果平台要以透明的方式执行存储的同步，则需要平台负责在 MIDlet 和同步引擎之间强制执行对存储记录的独占访问，这就影响了存储的效率。

记录存储中使用了一个长整形来表示日期和时间，记录它最后的更改时间。记录存储还维护一个整数版本号，应用程序中每更改一次记录存储的内容，版本号就会增加，这对同步引擎以及多个应用程序共用一个记录存储是十分有用。

记录是以字节数组的方式存放的。开发者可以使用 DataInputStream 和 DataOutputStream 以及 ByteArrayInputStream 和 ByteArrayOutputStream 成对组合和拆解不同的数据类型到字节数组中。每条记录通过一个整数记录 ID 号唯一标识，并且用作记录的主键。记录存储中生成的第一个记录号为 1，以后每增加一个记录号加 1。

6.1.2 记录存储 API

要使用记录存储功能，需要在应用程序中引入包：javax.microedition.rms。RecordStore 类是这个包的核心，它用来表示记录存储，以及相应的记录存储的生成、管理，并提供相应的方法进行记录的增加、删除和修改。如表 8-1 所示。

表 8-1 RecordStore 类方法说明

方法名称	方法原型和说明
addRecord	int addRecord(byte[] data,int offset,int numBytes); 增加新的记录
addRecordListener	void addRecordListener(RecordListener listener); 增加记录监听器
closeRecordStore	void closeRecordStore(); 关闭记录存储系统
deleteRecord	void deleteRecord(int recordId); 删除指定 ID 记录
deleteRecordStore	static void deleteRecordStore(String recordStoreName); 删除指定名称记录存储
enumerateRecords	RecordEnumeration enumerateRecords(RecordFilter filter,RecordComparator,boolean keepUpdated); 以可选指定的过滤器和比较器返回遍历记录的枚举器
getLastModified	long getLastModified(); 返回上次更改记录的时间
getName	String getName(); 返回记录存储的名字
getNumRecords	int getNumRecords(); 返回当前的记录条数
getNextRecordID	int getNextRecordID(); 返回下一条增加记录的记录 ID
getRecord	byte[] getRecord(int recordId); int getRecord(int recordId,byte[] data,int offset); 获得指定 ID 的记录条目数据

getRecordSize	int getRecordSize(int recordId); 获得指定 ID 的记录的以字节计数的大小
getSize	int getSize(); 返回当前记录存储的大小
getSizeAvailable	int getSizeAvailable(); 返回记录存储系统可用空间大小
getVersion	int getVesion(); 返回记录存储的版本号，此版本号每次记录存储更改都会赠家
listRecordStores	static String[] listRecordStores(); 返回当前 MIDlet 集所拥有的所有记录存储名称
openRecordStore	static RecordStore openRecord(String recordStoreName,boolean createIfNecessary); static RecordStore openRecord(String recordStoreName,boolean createIfNecessary,int authMode,boolean writable); 以指定名称打开或创建记录存储
removeRecordListener	void removeRecordListener(RecordListener listener); 删除指定的监听器
setMode	void setMode(int authMode,boolean writable); 改变当前记录存储的权限
setRecord	void setRecord(int recordId,byte[] newData,int offset,int numBytes); 重新设定指定 ID 记录的数据

这个包还包括以下 4 个接口，用来在记录存储操作中提供更多的功能。

- RecordComparator：用来比较记录存储中的两个记录。
- RecordEnumeration：提供记录存储双向索引的功能。
- RecordFilter：定义一个过滤器，是应用程序能根据自定义的标准匹配记录。
- RecordListener：用户监听记录存储的事件，比如增加、修改和删除。

为了保证记录存储操作的可靠性，RMS 包还提供了以下五个异常类，记录存储操作中可能遇到的异常情况。

- RecordStoreException：抛出一个记录存储的常规异常。下面 4 个异常从此异常派生，为记录存储异常的细化。
- InvalidRecordIDException：记录 ID 非法时抛出。
- RecordStoreFullException：纪录存储文件系统已满抛出。
- RecordStoreNotFoundException：指定名称的记录存储不存在时抛出。
- RecordStoreNotOpenException：所操作的记录存储没有打开的情况下抛出。

6.2 记录存储的管理

上文讲述了关于记录存储系统的基本原理和 API 的分层结构，下面讲述具体 API 的使用和注意点。

6.2.1 创建记录存储

记录存储类 RecordStore 的构造函数标记为私有，所以只能使用此类提供的两个静态成员函数来创建，这两个成员函数的如下所示：

```
static RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary, int authmode, boolean writable);
```

```
static RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary);
```

这里详细解释第一个成员函数的参数：

- recordStoreName：在此 MIDlet 组唯一记录存储名，由 1 到 32 个 Unicode 字符组成；
- createIfNecessary：值为 true 的时候，如果记录不存在的时候创建记录，否则返回记录。值为 false 的时候如果记录不存在的时候抛出 RecordStoreNotFoundException 异常，存在的时候返回记录。
- authmode：创建记录后是否允许其他 MIDlet 组存取，当值为 AUTHMODE_PRIVATE 时，只允许创建此记录的 MIDlet 组存取，当值为 AUTHMODE_ANY 允许任何 MIDlet 组存取。使用此参数时应该特别注意，一旦授予其他组可以写权限的时候，程序必须判断记录是否已经被其他 MIDlet 改写。
- writable：是否授予其他 MIDlet 组写记录的权限，这个参数只有当 authmode 为 AUTHMODE_ANY 是才有效，并且只有记录已经存在的时候 hulu 此参数。以后可以使用 setMode 更改。

第二个成员函数只是第一个成员函数以如下参数调用：

```
openRecordStore(recordStoreName, createIfNecessary, AUTHMODE_PRIVATE,false);
```

使用此成员函数的时候必须处理如下 3 个异常之一：

- RecordStoreException：记录存储系统相关的异常；
- RecordStoreNotFoundException：如果指定的记录不存在；
- RecordStoreFullException：记录存储系统已满；

此外如果 authmode 和 recordStoreName 错误的时候会抛出 IllegalArgumentException 异常。

按照如下的方式使用此函数：

```
RecordStore recordStore;
try
{
    recordStore = RecordStore.openRecordStore("test",true,RecordStore.AUTHMODE_PRIVATE,true);
} //以参数 createIfNecessary=true 调用的时候不抛出 RecordStoreNotFoundException 异常。
catch(RecordStoreFullException e) //存储系统已满异常
{
    System.out.println(e);
}
catch(RecordStoreException e) //存储系统异常
{
    System.out.println(e);
}
catch(IllegalArgumentException e) //参数错误异常
{
    System.out.println(e);
} //可以只处理一个异常 catch(Exception e){}。这样打印异常堆栈也可以知道具体的异常。
```

6.2.2 关闭和删除记录存储

记录打开后必须要关闭，如果不关闭，在程序退出的时候可能会发生数据并没有实际写入数据库的情况。打开和关闭数据库的次数要保持平衡。函数如下所示：

```
void closeRecordStore();
```

此函数很简单，没有参数，在一个存储记录实例上调用。如果关闭没有打开的记录会抛出 `RecordStoreNotOpenException`。当记录关闭的时候，所有的与此记录有关的 `listeners` 会被删除，`RecordEnumerations` 都将会失效。如果继续使用将会抛出 `RecordStoreNotOpenException` 异常。

数据记录创建后，也可以被删除，但是必须由创建此记录的 `MIDlet` 集才能够删除，并且是当此记录的所有打开的引用都已经关闭的情况下（包括本 `MIDlet` 集和被授权的其他 `MIDlet`）才能够删除。如果还有打开的记录，将会抛出 `RecordStoreException` 异常。此函数如下所示：

```
static void deleteRecordStore(String recordStoreName);
```

此函数为静态成员函数，可以删除本 `MIDlet` 集创建的名为 `recordStoreName` 的记录。如果此记录不存在，抛出 `RecordStoreNotFoundException` 异常。

这两个函数的使用如下所示：

```
try
{
    recordStore.closeRecordStore(); //关闭打开的纪录
    RecordStore.deleteRecordStore("test"); //删除前面创建的名为 test 的记录
}
catch(RecordStoreNotOpenException e) //记录没有打开的情况下关闭
{
    System.out.println(e);
}
catch(RecordStoreException e) //记录没有打开的情况下删除及其他存储系统异常
{
    System.out.println(e);
}
```

6.2.3 增加记录存储

记录存储系统中增加的记录条目是以二进制字节的方式加入的，相应的数据格式必须转换为二进制方式才能够存入记录。必须确保记录已经打开的情况下才能够增加条目。

函数如下所示：

```
int addRecord(byte[]data,int offset,int numBytes);
```

- `Data`：字节数组，可以为空。
- `offset`：字节数组中数据的起始位置。
- `numBytes`：字节数组中数据的长度。

返回值为当前新增加记录的 ID。

注意：增加记录操作为原子操作，必须等待记录增加完毕函数才能够返回。

由于记录条目是以二进制字节的方式加入的，所以首先必须把要增加的记录首先转换为二进制字节数组。下面是存储方式：

```
int addRecord(int coord) //在本例中，记录只使用简单的整型数据
{
    ByteArrayOutputStream baos=new ByteArrayOutputStream(); //创建字节流
    DataOutputStream outputStream=new DataOutputStream(baos); //创建数据输出流
```

```

int id = -1;
try{
    outputStream.writeInt(coord);           //把要存储的整数写入流
}
catch(IOException ioe)                     //处理 IO 异常
{
    System.out.println(ioe);
}
byte[] b=baos.toByteArray();              //转换为字节数组
try
{
    id = recordStore.addRecord(b,0,b.length);//增加记录条目到记录中
}
catch(RecordStoreException rse)           //捕获条目增加的异常（没有细分异常）
{
    System.out.println(rse);
}
try
{
    outputStream.close();                  //关闭输出流
    baos.close();                          //关闭字节流，必须以创建的顺序相反顺序关闭
}
catch(IOException ios)                     //捕获关闭流时的异常
{
    System.out.println(ios);
}
return id;
}

```

建议：增加记录时使用 `getSizeAvailable` 查询一下可用空间是否充足。

6.2.4 查询记录存储

如果不使用存储系统提供的 `RecordEnumeration` 接口，并且实现 `RecordComparator` 接口，系统提供的查询方式非常单一，必须提供存储的时候得到的记录条目 ID 才能够查询记录。如下所示：

```

byte[] getRecord(int recorded);
int getRecord(int record,byte[] buffer,int offset);

```

第一个函数是返回记录条目的字节数组。第二个函数从 `offset` 开始填充 `buffer`，返回填充的字节数，必须要注意从 `offset` 开始的 `buffer` 必须大于记录条目的字节数！可以使用如下的方法查询记录条目的字节数：

```

int getRecordSize(int recorded);

```

使用的时候必须确保 `recordId` 有效，可以使用下面两个方法来确保 ID 有效。

```

int getNumRecords();           //得到当前的记录条目总数
int getNextRecordID();         //得到下一条记录条目的 ID

```

注意：这里需要特别指出的是，记录存储系统增加的记录条目是从 1 开始的，与 Java 中的数组从 0 开始的不同。查询的时候一定要注意！

这里的查询结构为字节数组，还转换为程序中需要的数据。必须要根据存入的顺序来转换数据，如果顺序错误，转换结果将会出现错误。

```
int readRecord(byte[] data) //传入参数为获得的记录条目字节数组
{
    int coord=-1; //初始化数据，防止出现转换错误
    ByteArrayInputStream bais=new ByteArrayInputStream(data); //创建字节数组输入流
    DataInputStream inputStream=new DataInputStream(bais); //创建数据输入流
    try
    {
        coord=inputStream.readInt(); //从流中读取数据，必须按照写入的顺序读取
        inputStream.close(); //关闭数据输入流
        bais.close(); //关闭字节输入流，必须按照创建顺序反序关闭
    }
    catch(IOException e) //为了简单起见，捕获一种异常
    {
        System.out.println(e);
    }
    return coord; //返回转换的数据
}
```

6.2.5 修改记录存储

记录条目的修改也是相当的繁琐，如果没有使用提供的接口，必须要知道修改条目的 ID。这样才能够修改。

```
void setRecord(int recordId,byte[] newData,int offset,int numBytes);
```

除了第一个参数为记录 ID 外，其余几个参数含义如下：

- newData：新的记录条目数据；
- offset：数据的起始位置；
- numBytes：从起始位置开始的字节数；

必须确保记录 ID 有效，记录更改后立刻生效，再次查询的时候，将会得到新的记录条目数据。

```
try
{
    open(); //创建并打开记录
    addRecord(9); //增加一条记录
    form.addline("record 1 is "+readRecord(recordStore.getRecord(1))); //显示增加的记录
    byte[] data = recordStore.getRecord(1); //得到记录条目 1 数据
    data[3] = 10;
    data[2] = 2; //修改数据
    recordStore.setRecord(1,data,0,data.length); //修改记录条目 1 数据
    form.addline("after modify:record 1 is "+readRecord(recordStore.getRecord(1))); //显示修改的记录
    close(); //关闭记录
}
catch(Exception e) //为了简单起见，捕获单一异常
{
    System.out.println(e);
}
```

图 6-1 为程序运行效果。



图 6-1 修改记录条目演示

6.3 面对记录的高级操作

上文讲述查询记录条目的方式非常单一，而且效果有限，如果想要查询任意记录，基本上方法只有遍历记录集了。但是 RMS 同样提供了可以方便查询和监听的记录集的接口。

6.3.1 记录枚举接口

如果只能够使用记录条目 ID 来进行记录的查询，手段也太过单一，而且有可能因为 ID 的错误而抛出异常。可以使用 RecordEnumeration 接口来查询记录集中的数据，此接口不用用户自行实现而使用如下方法生成：

```
RecordEnumeration enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean
keepUpdated);
```

- filter：实现过滤器接口的类；
- comparator：实现比较器接口的类；
- keepUpdated：当其引用改变记录的时候，枚举器是否保持同步更新，如果保持同步更新，可能会损失性能，但是如果不保持同步更新，当其他引用更新记录以后，enumeration 可能会不正确，所以如果此参数设置为 false 的时候，需要特别的注意。

以指定参数条件调用此函数后生成一个可以根据指定条件遍历记录的枚举器。filter 和 comparator，都可以为 null，如果都为空，则枚举器会以未定义的顺序来遍历记录。经过测试，就是记录的逆序遍历。使用方法如下：

```
try
{
    open(); //打开记录
    addRecord(9);
    addRecord(7);
    addRecord(5);
    addRecord(6);
    addRecord(8);
    addRecord(4);
    addRecord(1);
    addRecord(2);
    addRecord(3); //增加 9 条记录
}
```

```

RecordEnumeration iter = recordStore.enumerateRecords(null,null,true);//以默认顺序遍历记录
while(iter.hasNextElement())           //是否有下一条记录
{
    form.append(readRecord(iter.nextRecord())+",") //读取下一条记录的数据
}
close();
} //关闭记录
catch(Exception e) //为了简单，只捕捉一种异常
{
    System.out.println(e);
}

```

这里演示了最常用的两个枚举器接口方法，`hasNextElement` 和 `nextRecord`。关于枚举器接口的其他方法，请参考 MIDP2.0 API。

6.3.2 记录过滤接口

记录过滤接口十分简单，只有一条函数。

```
boolean matches(byte[] candidate);
```

如果 `candidate` 匹配函数中的要求，返回 `true`，不匹配返回 `false`。可以用来过滤记录集中的数据。图 6-2 为默认的遍历顺序效果，图 6-3 为使用过滤器后的效果。



图 6-2 默认顺序遍历记录演示

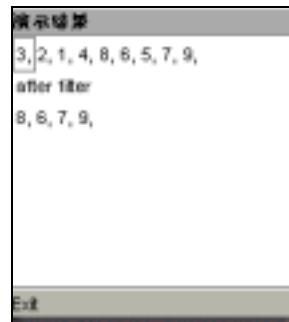


图 6-3 应用过滤器后遍历记录演示

```

class MoreFilter implements RecordFilter //此类实现大于给定数的过滤器
{
    private int criterion; //过滤标准
    public MoreFilter(int num) //过滤器构造函数，传入比较标准
    {
        criterion = num;
    }
    public boolean matches(byte[] candidate) //实现过滤器
    {
        if(readRecord(candidate)>criterion) //如果大于标准，返回 true
        {
            return true;
        }
        else
        {
            return false; //否则返回 false
        }
    }
}

```

```

    }
}
int readRecord(byte[] data)//此方法的实现，参考上文 6.2.4 的同名实现
}

```

上面是过滤器类的参考实现，实现了一个过滤掉小于给定值的过滤器。下面是如何使用过滤器：

```

//关于记录中的数据，使用上文 6.3.1 中的数据
RecordEnumeration iter=recordStore.enumerateRecords(null,null,true);
while(iter.hasNextElement())
{
    form.append(readRecord(iter.nextRecord()+"");
} //以默认顺序遍历记录
form.addline("");
form.addline("after filter");
iter = recordStore.enumerateRecords(new MoreFilter(5),null,true); //应用过滤器后得到的枚举器，
过滤掉小于 6 的值
while(iter.hasNextElement())
{
    form.append(readRecord(iter.nextRecord()+"");
} //应用过滤器后的遍历

```

关于过滤器的使用，还有一种方法，就是精确定义过滤器的条件，这样可以查找单一记录。

6.3.2 记录比较接口

应用了过滤器后的记录集，已经能够按照需要过滤掉无用的数据。还有几个接口，可以以指定的顺序来遍历记录，比较接口只有一个方法和 3 个静态的属性。

```
int compare(byte[] rec1,byte[] rec2);
```

此方法比较 rec1 和 rec2 的数据，决定 rec1 和 rec2 的数据代表的记录条目顺序。返回值为 3 个静态属性之一，如果 rec1 在 rec2 之前返回 PRECEDES，如果 rec1 在 rec2 之后返回 FOLLOWS，如果 rec1 和 rec2 没有先后顺序返回 EQUIVALENT。

下面是两个实现比较接口的比较器类的实现：

```

class SortAscend implements RecordComparator //以升序排列数据的比较器类
{
    public int compare(byte[]rec1,byte[]rec2)
    {
        int num1 = readRecord(rec1);
        int num2 = readRecord(rec2); //转换记录中的数据为需要的格式
        if(num1>num2)
        {
            return this.FOLLOWS; //num1 大于 num2 升序排列在 num1 在后
        }
        else if(num1<num2)
        {
            return this.PRECEDES; //num1 小于 num2 升序排列 num1 在前
        }
        else
        {

```

```

        return this.EQUIVALENT;    //num1 和 num2 相等，顺序也相等
    }
}
int readRecord(byte[] data);      //此成员函数的实现参考上文实现
}
class SortDescend implements RecordComparator //以降序排列数据的比较器类
{
    public int compare(byte[] rec1,byte[] rec2)
    {
        int num1=readRecord(rec1);
        int num2=readRecord(rec2);    //转换记录中的数据为需要的格式
        if(num1>num2)
        {
            return this.PRECEDES;    //num1 大于 num2 降序排列 num1 在前
        }
        else if(num1<num2)
        {
            return this.FOLLOWS;    //num1 小于 num2 降序排列 num1 在后
        }
        else
        {
            return this.EQUIVALENT;    //num1 和 num2 相等，顺序也相等
        }
    }
    int readRecord(byte[] data)      //参考上文 6.2.4 的实现
}

```

分别实现了升序和降序排列的比较器，下面演示比较器如何使用：

```

//关于记录中的数据，使用上文 6.3.1 中的数据
RecordEnumeration iter=recordStore.enumerateRecords(null,null,true);    //正常顺序生成枚举器
while(iter.hasNextElement())
{
    form.append(readRecord(iter.nextRecord())+",");
}
form.addline("");
iter = recordStore.enumerateRecords(null,new SortAscend(),true);    //使用升序比较器生成枚举器
form.append("ascend sort:");
while(iter.hasNextElement())
{
    form.append(readRecord(iter.nextRecord())+",");    //升序遍历
}
form.addline("");
iter = recordStore.enumerateRecords(null,new SortDescend(),true);    //使用降序比较器生成枚举器
form.append("descend sort:");
while(iter.hasNextElement())
{
    form.append(readRecord(iter.nextRecord())+",");    //降序遍历
}
form.addline("");

```

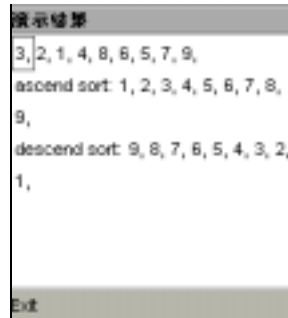


图 6-4 使用升序和降序比较器遍历记录演示

6.3.3 记录监听接口

记录存储系统提供了一种方法可以监听记录的变化，一旦记录发生变化，将会触发相应得函数，这个时候用户可以执行相应的操作。主要包括以下三个方法：

```
void recordAdded(RecordStore recordStore,int recordId);
void recordChanged(RecordStore recordStore,int recordId);
void recordDeleted(RecordStore recordStore,int recordId);
```

这几个方法的第一个参数都是发生变化的记录的引用，第二个参数是发生变化的记录的 ID。第一个方法是增加记录的时候触发，recordId 为增加记录的 Id；第二个方法是有记录改变的时候触发；第三个方法为记录删除时触发。记录删除后，此 ID 将不再有效。

建议：这几个被触发的方法中，建议不要在其中增加或者删除修改记录，这样会造成循环触发。

```
class RecordListenerDemo implements RecordListener
{
    public void recordAdded(RecordStore recordStore,int recordId) //实现增加记录监听方法
    {
        try
        {
            System.out.print("Added Id is: "+recordId);           //打印增加记录的 ID
            System.out.println(" Data is: "+readRecord(recordStore.getRecord(recordId)));//打印增加的
记录
        }
        catch(Exception e)                                       //由于获取记录，必须捕获异常
        {
            System.out.println(e);
        }
    }
    public void recordChanged(RecordStore recordStore,int recordId) //实现改变记录监听方法
    {
        try
        {
            System.out.print("Changed Id is: "+recordId);       //打印改变记录的 ID
            System.out.println(" Data is: "+readRecord(recordStore.getRecord(recordId)));//打印改变的
记录
        }
        catch(Exception e)                                       //由于获取记录，必须捕获异常
```

```

        {
            System.out.println(e);
        }
    }
    public void recordDeleted(RecordStore recordStore,int recordId) //实现删除记录监听方法
    {
        System.out.println("delete Id is: "+recordId);           //打印删除记录 ID，不能使用获取记
录值
    }
    int readRecord(byte[] data)                                //参考上文 6.2.4 的实现
}

```

上面的类简单的演示了当记录改变的时候改变的记录的 Id 或内容，没有实现有意义的工作。按照如下的方式使用时：

```

listener = new RecordListenerDemo();           //构造监听者
recordStore.addRecordListener(new RecordListenerDemo(listener)); //增加监听者
recordStore.addRecord(9);                       //增加记录
byte[] data = recordStore.getRecord(1);         //获取记录条目数据
data[3] = 10;                                   //改变数据值
recordStore.setRecord(1,data,0,data.length);   //改变记录条目值
recordStore.deleteRecord(1);                   //删除记录条目
recordStore.removeRecordListener(listener);     //删除监听者

```

产生如下的输出：

```

Added Id is: 1 Data is: 9
Changed Id is: 1 Data is: 10
delete Id is: 1

```

必须首先使用 `addRecordListener` 增加监听者，可以增加多个实现监听接口的侦听器。可以使用 `removeRecordListener` 来删除相应的监听者。

6.4 存储记录的格式问题

记录存储的实际格式为二进制数据，无论读取或者写入数据的时候，都必须使用二进制数据，实际表现就是必须转换为字节数组。这样使用起来相当繁琐。不知道读者有没有注意到，上文中的枚举器使用的时候有点效率的问题，就是无论使用过滤器或者是比较器，都必须把二进制数据转换位需要的数据格式，加上过滤和比较，很多地方都重复转换了多次，如果数据量庞大，操作在时间花销会相当可观。特别是对于手机游戏这种需要反应迅速的程序，此种使用方式基本上没有多少实际的用处。

6.4.1 二进制格式和其他格式的转换

二进制数据格式和其他数据格式之间的转换主要使用如下 4 个类：

- `ByteArrayInputStream`：使用字节数组来构造；
- `ByteArrayOutputStream`：来生成数据的字节数组，也就是二进制数组；
- `DataInputStream`：用来读取 `ByteArrayInputStream` 中的数据；
- `DataOutputStream`：数据写入此流，然后根据流中的数据生成 `ByteArrayOutputStream`；

上面的 4 个类必须配对使用,就是说 `ByteArrayInputStream` 与 `DataInputStream` 一起使用, `ByteArrayOutputStream` 与 `DataOutputStream` 一起使用。

`DataInputStream` 和 `DataOutputStream` 中包含了丰富的方法来进行 Java 中所有的基础数据类型转换为数据流格式。比如说 `int`, `String`, `char` 等数据类型,但是不包括引用和类,如果要是想转换为二进制格式,必须在类中提供方法把类中的需要保存基础数据类型数据写入流,并且提供方法可以把流中的数据重新写回类。MIDP 中不包括序列化操作。所以必须手工来做。可以按照下面的方式来做:

```
class Test //使用方法演示类
{
    public int x; //基础数据类型
    public int y; //基础数据类型
    public void writeData(DataOutputStream dos) throws IOException //写入数据到流中
    {
        dos.writeInt(x);
        dos.writeInt(y); //注意此处写入的顺序
    }
    public void readData(DataInputStream dis) throws IOException
    { //从流中读取数据
        x = dis.readInt();
        y = dis.readInt(); //读取的时候,必须与写入的顺序相同
    }
}
```

6.4.2 几个关于数据转换的问题

- 游戏中的数据这里需要特别指出的是,只保存能够恢复游戏需要的数据。而且由于游戏中的可能需要存储的数据量偏大,数据类型需要存储尽可能小的数据类型,可以在存取的时候首先转换为占用空间较小的类型,这样读取的时候直接赋值就可以了。
- 对于移动设备来说,它的字符串一般使用 UTF8 格式,所以对于 `String` 类型,应该使用 `writeUTF` 和 `readUTF` 方法。

6.5 游戏中的数据存储

如果使用记录存储系统提供的手段来进行记录的存储和排序,效率将会十分的低效。基本上不能满足实时数据存储的要求。下文将会介绍一种方法来基本上可以满足实时数据存储的要求。

6.5.1 游戏中记录存储的方式

本文将会介绍一种方式,可以使用在游戏的记录存储中。按照通常每个数据一条记录的存取方式,既占用空间,而且在存取的时候还占用时间。这里提出一种方案,不使用记录存储系统的其他功能,只使用记录系统的第一条记录,无论修改或者删除数据,都是在在这一

条记录中进行，这样就解决了存储空间和存取时间的问题。下面提供了一个按照此种方式使用的类：

```

import javax.microedition.rms.*;
public class LPRecordStore //游戏记录存储类名
{
    private RecordStore rs; //系统的记录存储类
    private String rsName; //使用的记录存储名
    public LPRecordStore(String recordName) //构造函数，使用记录名来构造
    {
        rsName=recordName; //存储记录名，删除记录时使用
        try
        {
            rs=RecordStore.openRecordStore(rsName,true);//以指定的记录创建或者打开记录
        }
        catch(Exception e)
        {
            System.out.println(e); //为了简单起见，只捕获最简单的异常，下文同
            //控制台打印输出
        }
    }
    public void close() //关闭记录，使用完毕必须关闭记录
    {
        try
        {
            rs.closeRecordStore();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
    public boolean empty() //查询记录中是否有数据
    {
        try
        {
            if(rs.getNumRecords()>0) //得到记录的数据数
            {
                return false; //如果不为零返回 false
            }
            else
            {
                return true; //如果记录数为 0，返回 true
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        return true; //如果出现异常，返回记录为空
    }
    public void setRecord(byte[] buffer) //透明的设置数据，不管数据是新增加数据或者是修改

```

数据

```

    {
        try
        {
            if(empty())
            {
                rs.addRecord(buffer,0,buffer.length);//如果记录为空，增加记录条目
            }
            else
            {
                rs.setRecord(1,buffer,0,buffer.length);//如果不为空，修改第一条记录
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
    public byte[] getRecord()                //得到记录的二进制字节数组
    {
        byte buffer[];
        try
        {
            buffer=rs.getRecord(1);
            return buffer;
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        return null;                        //如果记录中没有数据或者出现异常，返回为 null
    }
    public void deleteRecord()              //删除记录，这样可以直接删除数据库，直到使用的时候再
    创建
    {
        try
        {
            rs.deleteRecordStore(rsName);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

```

上面的类可以使记录把存储系统当作二进制文件系统来使用，就是可以读取写入数据，唯一不同的是，写入的时候需要整个“文件”写入，读取的时候把整个“文件”读出来。

6.5.2 游戏记录存储类的使用

下面演示上一节提供的类的使用：

```

try
{
    Test[] test = new Test[10];           //此处测试类的定义已经在上文提供
    ByteArrayOutputStream baos=new ByteArrayOutputStream(); //生成字节输出流
    DataOutputStream dos=new DataOutputStream(baos);       //数据输出流
    form.addline("存入的数据");
    for(int i = 0;i<test.length;i++)
    {
        test[i] = new Test();           //构造测试数据
        test[i].x = i;                 //横坐标为 0-9
        test[i].y = test.length-i;     //纵坐标为 9-0
        form.append(""+test[i].x+","+test[i].y+""); //显示数据
        test[i].writeData(dos);        //写入流
    }
    form.addline("");
    LRecordStore rs = new LRecordStore("lipeng");         //生成并打开记录
    rs.setRecord(baos.toByteArray());                    //保存记录数据
    rs.close();                                         //关闭记录
    rs = new LRecordStore("lipeng");                   //重新生成并打开记录
    byte[] data = rs.getRecord();                      //读取记录中的数据
    ByteArrayInputStream bais=new ByteArrayInputStream(data); //构造字节输入流
    DataInputStream dis=new DataInputStream(bais);     //数据输入流
    Test []test2 = new Test[10];                      //构造新的测试数据
    form.addline("读取的数据");
    for(int i = 0;i<test2.length;i++)
    {
        test2[i] = new Test();
        test2[i].readData(dis);                      //从流中读取数据
        form.append(""+test2[i].x+","+test2[i].y+""); //显示
    }
    form.addline("");
    rs.close();                                       //关闭记录
}
catch(Exception e)
{
    System.out.println(e);
}

```

图 6-5 是演示结果,可以看到完全恢复了当时保存的数据。而且上面演示了游戏中可能用到数据存储技术。存储能够完全恢复游戏中状态的数据,然后在重新载入游戏的时候,重新构造对象,并恢复数据。这里对象的构造函数可以增加一个可以读取输入流的构造函数。这样直接以输入流来构造就可以了。

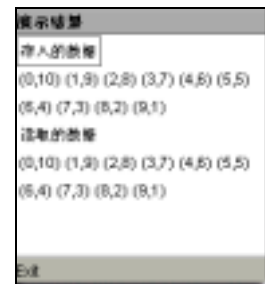


图 6-5 游戏记录存储类的效果演示

6.5.3 与传统的存取方式的对比

上节演示了游戏中的存取方式,这里与传统的存取方式作一对比,由于记录的存取时间需要在手机上进行测试,每款手机都不相同,这里提供的数据仅仅有比较参考意义。有意义的单个数据就是关于数据

存储大小的比较。

这里的传统数据存取方式可以就是把上节演示的数据，分 10 条存入记录。然后于上文的数据作出对比。下表数据为存取 10 条 test 数据为测试数据。

表 8-2 不同数据存储方式的比较

对比参数	使用系统提供的方式	使用本文提供的游戏存储方式
写入操作	266ms	16ms
读取操作	16ms	1ms
空间大小	368 字节	144 字节

从上表可以看出，使用本文提供的游戏数据存取方式，存取速度提高一个数量级以上，这个优势如果在数据量更大的时候，将会相当的客观。在存取数据大小方面，使用系统提供的方式是使用游戏数据存储方式大小的几乎 3 倍。

通过对比，可以看出系统提供的方式几乎没有使用的价值。建议使用游戏中的数据存取方式。而且对于使用游戏中的数据存取方式，还可以使用压缩存储的技术。这样对于数据量大的游戏，十分重要，而使用系统提供的方法，没有提供数据压缩方法。

6.6 本章小结

本章详细叙述了 MIDP 中提供的记录存储系统的使用及使用实例，而且提出了一个极为有价值的游戏中的数据存储方法和完整的例子。在本章的最后，给出了使用系统提供的技术和本文给出的方法的效率和空间对比。

本章的重点是对游戏中使用的数据存储方式的介绍，读者需要熟练使用数据的读取和写入技术。以及对标准二进制数据的存储方式。读者需要特别注意的是二进制数据的存储方式，在读写过程中，必须写入与读取的顺序相同，这样才能保证数据存储的正确。