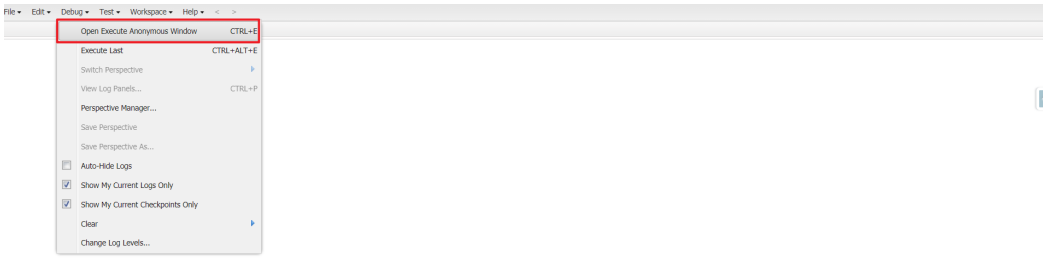
Salesforce Apex 零基础入门

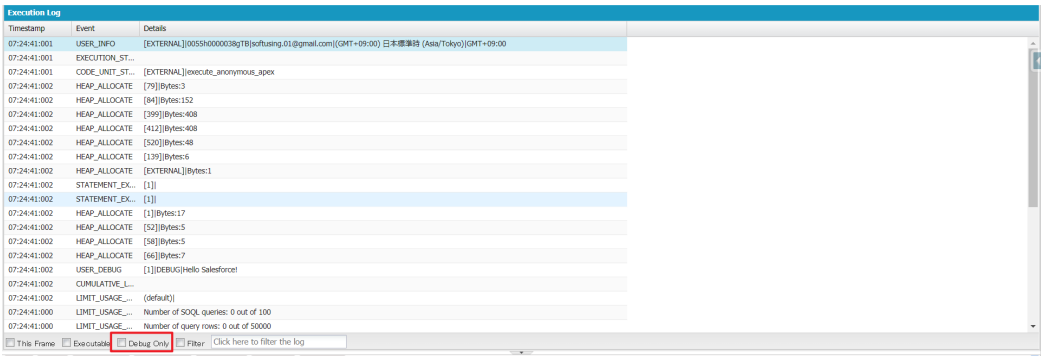
第1章 Hello Salesforce!

```
// 第1条语句
System.debug('Hello Salesforce');
```

- System开头字母大小写都可以，建议开头字母大写
- 要输出的字符用单引号括起来
- 在行尾要添加一个分号，表示程序的结束



选中Open Log，然后Execute



第2章 变量及其命名规则

2.1 什么是变量

现实世界中，装水要有水桶，盛饭要用碗，在计算机的世界中，保存东西用的叫变量。

2.2 变量的命名规则

变量的名字可以随便起，比如说学生的学号姓名，

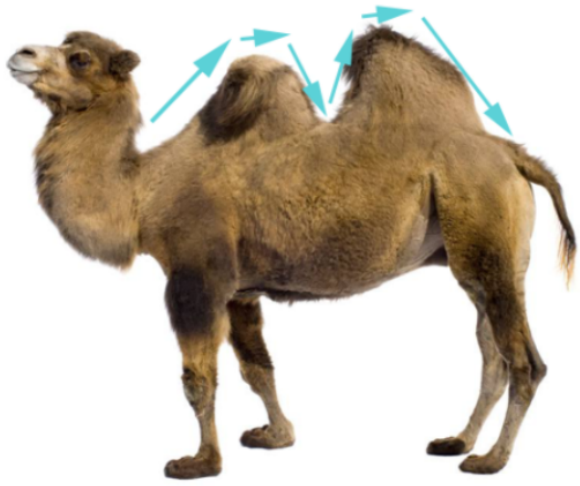
- studentNo ~~abc1123 = 55;~~
- studentName studentName = 'JZY';

```
studentNo = 38;
```

①**小写字母开头**，开头不能用下划线（_），~~*,?,<~~
②**驼峰命名法**

什么是驼峰命名法?

变量名的第一个单词的首写字母小写，后面的单词的第一个字母大写。



第3章 变量类型（Integer）

studentNumber就是一个变量名，不同类型的变量存储在计算机里的大小也是不一样的，一个数字的大小肯定和一大段文章，或者一段视频是不一样的，所以就有变量类型的概念

比如说我们要学的第一个变量类型Integer，就是一个整数类型。

```
Integer studentNumber = 55;
```

变量类型

当我写下Integer的时候，我就是在告诉计算机给我准备一个能存放整数类型的大小的空间，当然是放在内存里的 。

比如说年龄，学生的数量，就是个数字，学生的姓名就是个字符串（String）

```
Integer studentAge = 130;
System.debug('studentAge'); // 推荐
system.debug(studentAge);
```

- 加号起到拼接的作用
- 在程序中使用// 是注释的意思，为了加一些提示，里面的代码不执行
- +号表示连接

```
Integer studentAge = 130;
// 加号起到拼接的作用
// 学生的年龄是：39!
System.debug('学生的年龄是：' + studentAge);
```

第4章 关键字和保留字

```
// 有些单词背占用，不让我们使用
Integer system = 39;
Integer Integer = 39;

// 下面的是可以的
Integer integerAge = 39;
```

第5章 再说Integer

Integer文档

说起整数，一定要有大小，不能无限大，而且是有正负的

Integer最小值-2,147,483,648、最大值2,147,483,647

```
// Error 当是负的2147483648的时候，无法执行
// Integer i1 = -2147483648;
Integer i1 = -2147483647;
Integer i2 = 2147483647;
System.debug(i1);
System.debug(i2);
```

大家知道这两个数字的由来吗？

-2,147,483,648 = - 2^31;

2,147,483,647 = 2^31 - 1;

第6章 变量的声明和初始化

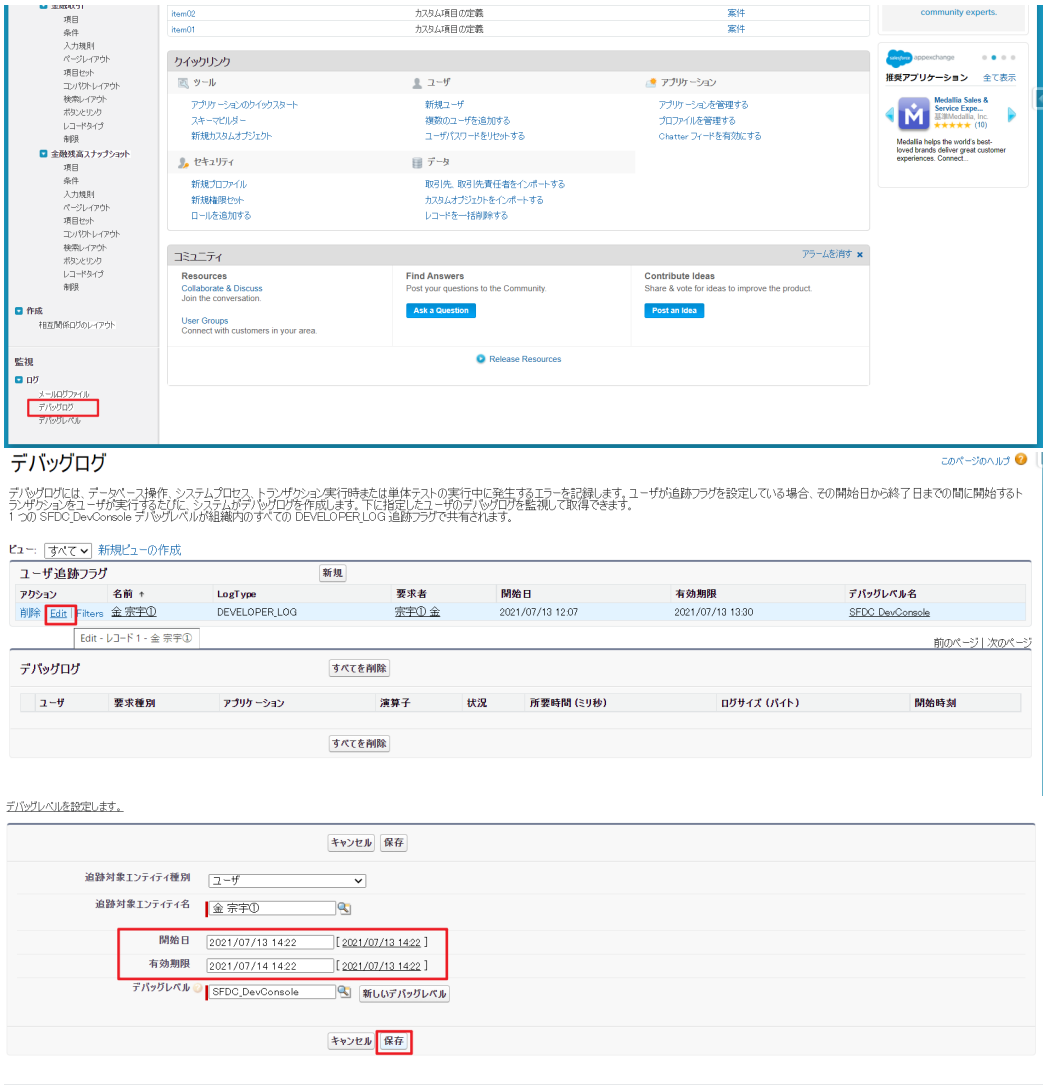
```
// 变量的声明
// 变量的初始化,就是给一个值
Integer i2 = 130;

// 变量的声明
Integer i1;

// 变量的初始化,就是给一个值
i1 = 130;
System.debug(i1);
```

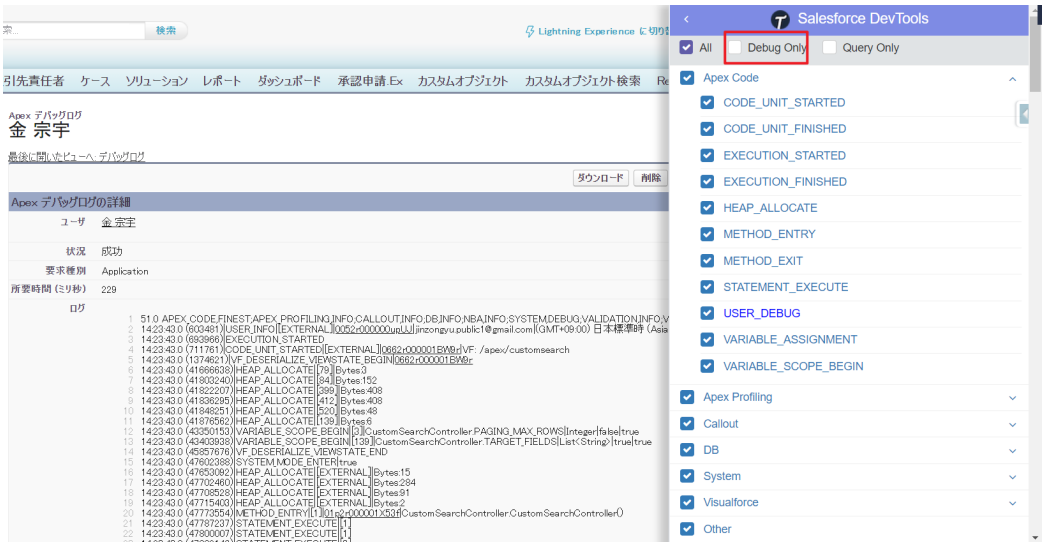
```
// 变量的声明
Integer i1;

// 在只声明，没有赋值的情况下，就是null
System.debug(i1);
```

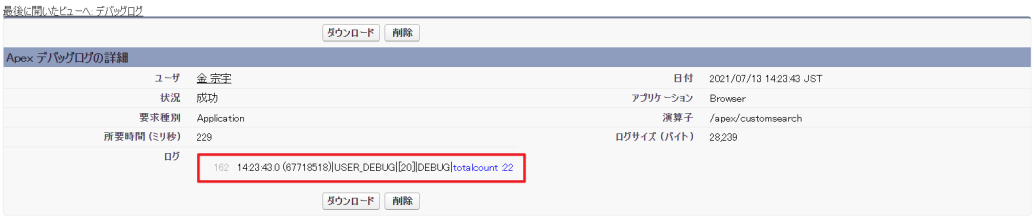
第7章 System.debug()实际应用

大多是在调试程序的时候使用，因为apex代码不像Java一样有很好的调试功能，因为Salesforce的代码在云上，所有有的使用为了定位问题，需要使用System.debug()

调试完程序可以删掉，也可以保留，只要不出log，对效率是不会有影响，但是推荐删除点，但是在测试的类里面无所谓。







第8章 基本数据类型

8.1 基本数据类型（11种）

---

①Integer（整数）

32 位整数。Integer最小值−2,147,483,648、最大值2,147,483,647 。

Integer 类

```
Integer age = 130;

```

---

②Long（长整数）

64位整数。Longs最小值是负的2的63次幂、最大值是2的63次幂−1。

比Integer更广泛的时候使用。

```
Long l = 2147483648L;
```

---

③Double 没有float

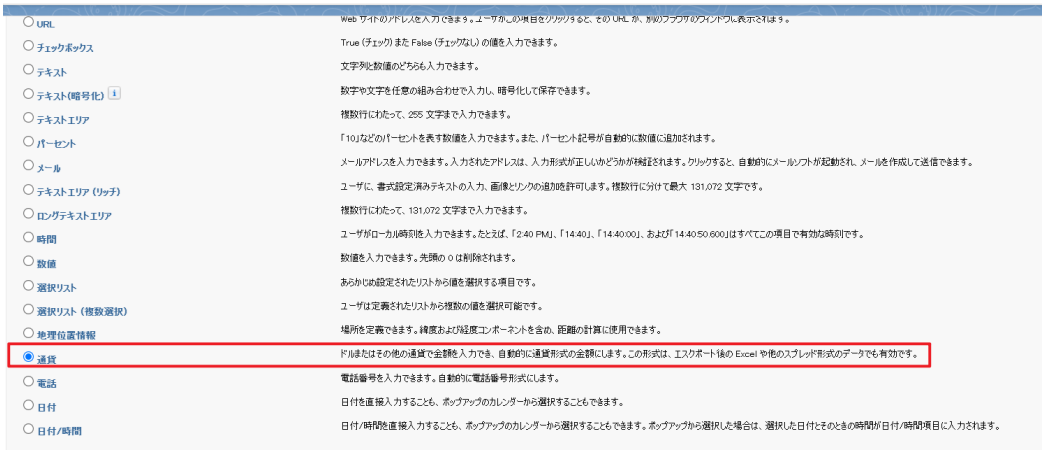
64位小数。Doubles最小值是负2的63次幂、最大值是正2的63次幂−1。

```
Double d=3.14159;
```

---

④Decimal

小数点を含む数値。Decimal は、任意の精度数です。**通貨項目には自動的に Decimal 型が割り当てられます。**



---

⑤String

使用单引号,比如说一段文字

String类型

```
String s = 'This is salesforce variable';

String s = 'Hello Maximillian';
String s2 = s.abbreviate(8);
System.debug('Hello...' + s2);
System.debug(s2.length());

String s = 'This is salesforce variable';
System.debug('s');
System.debug(s);
```

自己练习的时候：

- **assertEquals 换成debug**
- **第一个参数删掉**
- **参照下面的例子**

```
String s = 'Hello Maximillian';
String s2 = s.abbreviate(10);
System.assertEquals('Hello...', s2);
System.assertEquals(8, s2.length());

String s = 'Hello Maximillian';
String s2 = s.abbreviate(8);
System.debug(s2);
System.debug(s2.length());
```

---

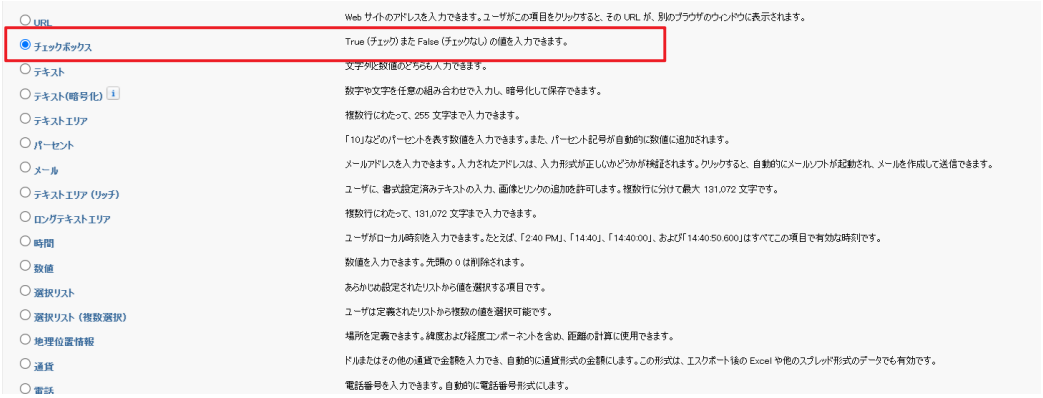
⑥Boolean

只有True和False两种值，比如说性别

在チェックボックス的时候用

```
Boolean b = Boolean.valueOf('true');
```

```
System.Debug(b);

Boolean b = true;
System.Debug(b);
```



---

⑦Blob

二进制数据，在处理文档的时候使用

Blob类型

---

⑧Date

Date Class

```
取得当前日期：
Date dt = Date.today();
System.debug(dt);

测试的时候经常用：
Date myDate = date.NewInstance(1960, 2, 17);
System.debug(myDate);

判断闰年：
System.debug(Date.isLeapYear(2004) ) ;

boolean bln1 = Date.isLeapYear(2004);
boolean bln2 = Date.isLeapYear(2021);

System.debug(bln1);
System.debug(bln2);
```

⑨Time

Time Class

```
Time myTime = Time.newInstance(18, 30, 2, 20);
System.debug(myTime);
```

---

⑩Datetime

CMT时间：格林威治时间

```
DateTime dt = DateTime.now();
System.debug(dt);

※因为标准的对象使用的是格林威治时间，自定义的对象也要使用格林威治时间
XXXDatetime__c = Datetime.now().addHours(9);
```

---

⑪ID

ID Class

Saleforce中每一条资源都是有ID的

资源包括：实际的数据，Metadata（profile，tab等等）

Classic 15位

Lightning 18位

```
Account a = new Account(name = 'account');
insert a;
Id myId = a.id;
system.debug(myId.getSobjectType());
```

8.2 集合类型

8.2.1 List

List Class

1.从第0条记录开始

2.存储某种类型的记录的集合

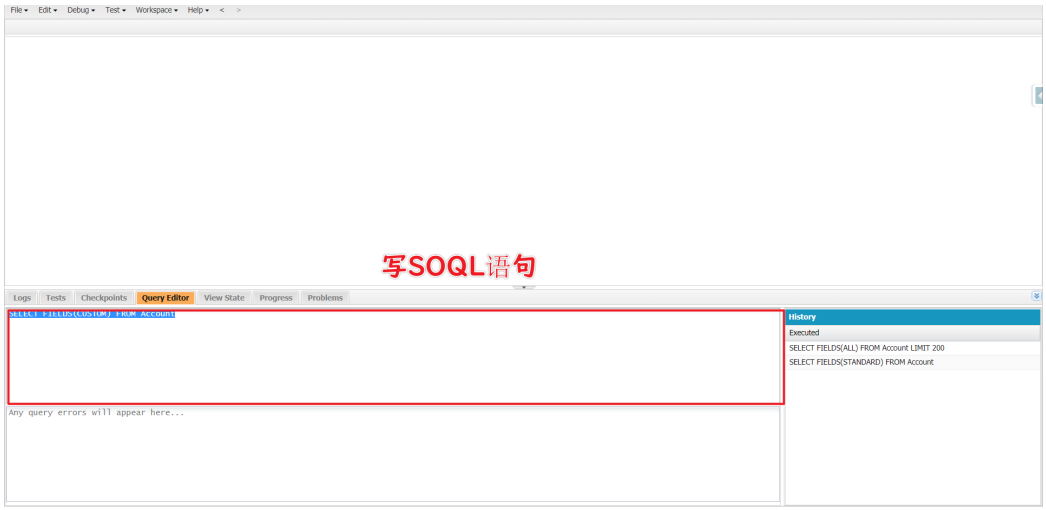
List<String>

String类型的集合

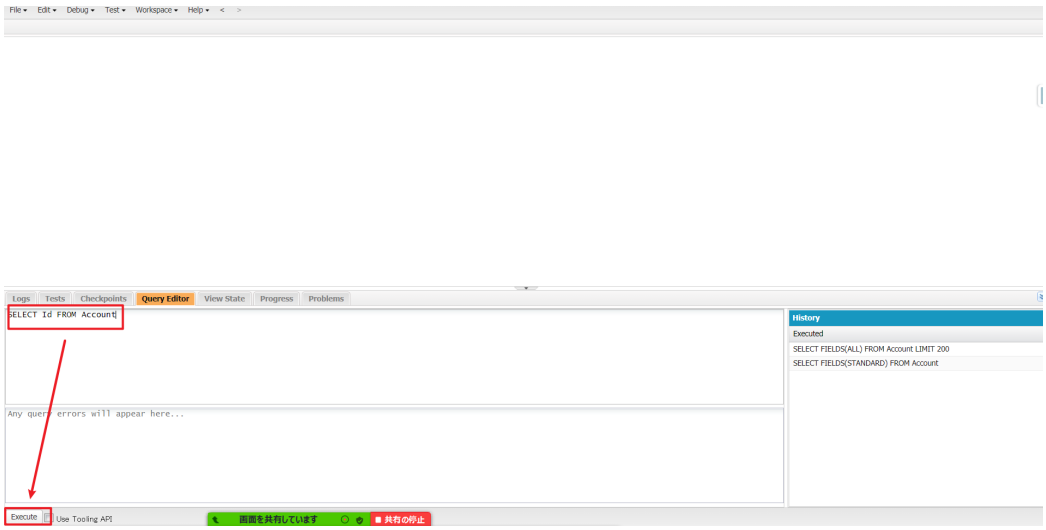
而且只能是同一种类型

Integer

String



SQOL:::: Salesforce Object Query Language (SOQL)



大小写无所谓，一般第一个字母，或者关键字整个大写

SELECT:选择　固定的

ID：你要选需求的值　可以添加对象里面存在的项目

From　固定的，从后面的对象里面来选择

Account：你的环境里面标准的对象，或者自定义的对象

```
SELECT Id FROM Account
SELECT Id, Name FROM Account order by id

如果没写order by，应该默认通过id排序
```

SOSL,DML

```
// list里放的是果篮
List<Account> accList = [SELECT Id, Name,MasterRecordId FROM Account];

// accList[0] : account
// MasterRecordId
System.debug(accList[0].Id);
System.debug(accList[0].Name);
```

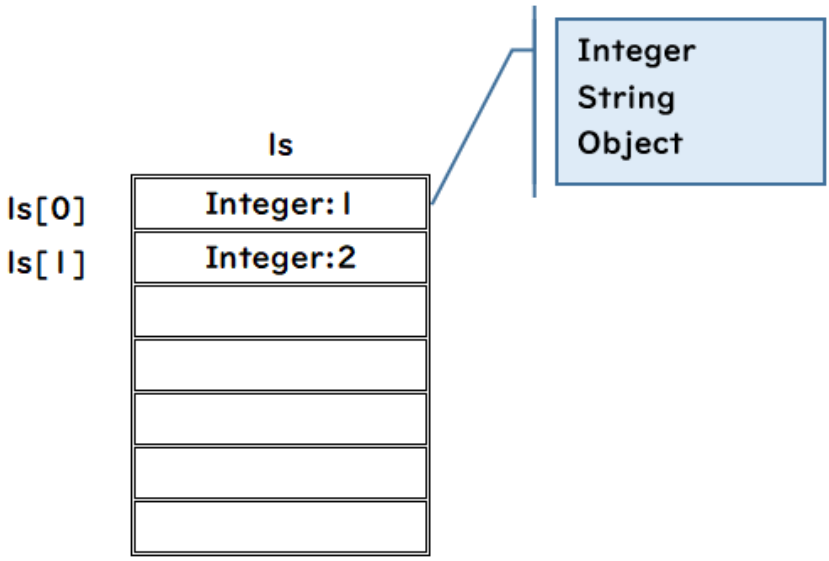
```

```

・创建・添加・取得・修改・删除

~~直接创建~~

[//] 两个反斜杠相当于注释

```
List<Integer> ls = new List<Integer>();
//
//List<String> ls = new List<String>();  // 前后必须保持一致
//List<Integer> ls = new List<String>(); // 错误

ls.add(1);
ls.add(2);
ls.add(3);
// 计算机里面第一个集合元素是0开头
System.debug(ls[0]); //取得第一个元素  ls.add(1);
System.debug(ls[1]); //取得第二个元素  ls.add(2);
System.debug(ls[2]); //取得第三个元素  ls.add(3);

// ls[1] = ls.get(1) 相同
System.debug('ls.get(1):' + ls.get(1)); //取得

```



2021/08/08 ⬇⬇⬇⬇⬇⬇⬇⬇⬇⬇

Salesforce删除权限

ユーザがレコードを削除するには、3 つのシナリオがあります:

- レコードを削除しようとしているユーザがレコードの所有者です。
- ユーザが所有者のロール階層において所有者よりも上位に位置します。
- レコードを削除しようとするユーザが「すべて変更」権限を持っています。

- 所有者可以删除，也就是Owner

- Role阶层上位的人可以删除下位
- Object的「すべて変更」权限有的情况

※主从关系(A是主,AA是子)，共有设定选主从联动，通过sharing rule共享了权限，共享先删不掉共享元的主对象的数据，但是可以删掉从对象的数据

关于上节课的SOQL语句的补充

- Limit 关键字后面接数字，表示显示的最大件数
- Order By后面接字段名字 默认是升序

```
ASC: ascending    升序 默认写不写都可以
DESC: descending  降序 需要写上

SELECT id,name,Date01__c FROM FormulaConfirm__c Order By Date01__c,Name
可以添加多个排序项目
```

- Group By 分组 （还有个Having过滤，以后再说）

1. 单个字段分组
2. 多个字段分组
3. 常用聚合函 **count() ， sum() ， avg() ， max() ， min()**

> Select YoteiDate__c,sum(Money__c),sum(Quantity__c) from SoqlTestObject__c Group By YoteiDate__c

```
Select YoteiDate__c,sum(Money__c),sum(Quantity__c)
from SoqlTestObject__c Group By YoteiDate__c

Select calendar_month(YoteiDate__c),sum(Money__c),sum(Quantity__c)
from SoqlTestObject__c Group By calendar_month(YoteiDate__c)

calendar_month 08
CALENDAR_YEAR  2021

SELECT CreatedDate, Amount
FROM Opportunity
WHERE CALENDAR_YEAR(CreatedDate) = 2021

CALENDAR_YEAR之类的函数尽量用在where条件里面
要想在选择项目中表示的情况下，后面也要使用
```

https://developer.salesforce.com/docs/atlas.ja-jp.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_select_date_functions.htm

Group By 后面的可以在显示的字段中表示，其他的一律不行

如果要显示其他字段，必须是加入集合函数**sum**

为了排除干扰，我们来创建一个对象 SoqlTestObject

予定日	YoteiDate	日付
予定金額	Money	通貨(18、0)
数量	Quantity	数値(18、0)
Dummy	Dummy	テキスト(80)

	SoqlTestObject名	予定金額	予定日	数量
1	A	¥100	2021/08/08	10
2	A	¥2,000	2021/08/08	20
3	B	¥2,001	2021/08/09	500
4	B	¥6,009	2021/08/10	123,213

```
SELECT Id FROM Account
SELECT Id FROM Account Limit 1

// 标准Object
SELECT FIELDS(ALL) FROM Account LIMIT 200
SELECT FIELDS(CUSTOM) FROM Account LIMIT 200
SELECT FIELDS(STANDARD) FROM Account

// 自定义Object
SELECT FIELDS(ALL) FROM FormulaConfirm__c LIMIT 200
SELECT FIELDS(CUSTOM) FROM FormulaConfirm__c LIMIT 200
SELECT FIELDS(STANDARD) FROM FormulaConfirm__c

当使用FIELDS关键字的时候， FIELDS(ALL)和FIELDS(CUSTOM) 的时候 最大是200
不适用FIELDS关键字的时候，不能超过50000，一次查询实际数据超过50000条，
就会触发ガバナー制限（后面的课程会有实例代码）
```

```
SELECT FIELDS(STANDARD) FROM FormulaConfirm__c
```

NO	項目の表示ラベル	API 参照名	データ型	桁数	項目タイプ	必須	選択リスト値
1	カスタムオブジェクト ID	Id	id		標準	必須	
2	所有者 ID	OwnerId	参照関係(小组,用户)		標準	必須	
3	削除	IsDeleted	チェックボックス		標準		
4	数式確認用オブジェクト名称	Name	テキスト	80	標準		
5	レコードタイプ ID	RecordTypeId	参照関係(记录类型)		標準		
6	作成日	CreatedDate	日付/時間		標準	必須	
7	作成者 ID	CreatedById	参照関係(用户)		標準	必須	
8	最終更新日	LastModifiedDate	日付/時間		標準	必須	
9	最終更新者 ID	LastModifiedById	参照関係(用户)		標準	必須	
10	System Modstamp	SystemModstamp	日付/時間		標準	必須	
11	最終閲覧日	LastViewedDate	日付/時間		標準		
12	最終参照日	LastReferencedDate	日付/時間		標準		
13	Formula01	Formula01_c	数式(テキスト)	1300	カスタム		
14	Product_type	Product_type_c	ロングテキストエリア	255	カスタム		
15	Formula02	Formula02_c	数式(テキスト)	1300	カスタム		
16	Days_Open	Days_Open_c	数値(18, 0)	18	カスタム		
17	Date01	Date01_c	日付		カスタム		
18	選択リスト	Field1_c	選択リスト		カスタム		AAA;BBB;CCC;DDD
19	連動1	Field2_c	選択リスト		カスタム		AAA;BBB;CCC;ABC
20	連動2	Field3_c	選択リスト		カスタム		AAA1;AAA2;AAA3;BBB1;BBB2;C
21	list1	list1_c	選択リスト		カスタム		AAA;BBB
22	list2	list2_c	選択リスト		カスタム		AAA1;AAA2;BBB1;BBB2

```
SELECT FIELDS(CUSTOM) FROM FormulaConfirm__c LIMIT 200
```

NO	項目の表示ラベル	API 参照名	データ型	桁数	項目タイプ	必須	選択リスト値
1	カスタムオブジェクト ID	Id	id		標準	必須	
2	所有者 ID	OwnerId	参照関係 (小组,用户)		標準	必須	
3	削除	IsDeleted	チェックボックス		標準		
4	数式確認用オブジェクト名称	Name	テキスト	80	標準		
5	レコードタイプ ID	RecordTypeId	参照関係 (记录类型)		標準		
6	作成日	CreatedDate	日付/時間		標準	必須	
7	作成者 ID	CreatedById	参照関係 (用户)		標準	必須	
8	最終更新日	LastModifiedDate	日付/時間		標準	必須	
9	最終更新者 ID	LastModifiedById	参照関係 (用户)		標準	必須	
10	System Modstamp	SystemModstamp	日付/時間		標準	必須	
11	最終閲覧日	LastViewedDate	日付/時間		標準		
12	最終参照日	LastReferencedDate	日付/時間		標準		
13	Formula01	Formula01_c	数式 (テキスト)	1300	カスタム		
14	Product_type	Product_type_c	ロングテキストエリア	255	カスタム		
15	Formula02	Formula02_c	数式 (テキスト)	1300	カスタム		
16	Days_Open	Days_Open_c	数値 (18, 0)	18	カスタム		
17	Date01	Date01_c	日付		カスタム		
18	選択リスト	Field1_c	選択リスト		カスタム		AAA;BBB;CCC;DDD
19	連動1	Field2_c	選択リスト		カスタム		AAA;BBB;CCC;ABC
20	連動2	Field3_c	選択リスト		カスタム		AAA1;AAA2;AAA3;BBB1;BBB2;CC
21	list1	list1_c	選択リスト		カスタム		AAA;BBB
22	list2	list2_c	選択リスト		カスタム		AAA1;AAA2;BBB1;BBB2

Where 语句

1. 字符串条件

```
Select id,name from SQOL__c where name = 'A'
Select id,name,Money__C from SQOL__c where Name = 'AA' and Money__C = 1
```

2. 各种符号

```
select id,name,Money__c from SQOL__c where Money__c > 1000 and Money__c < 3000
```

※没有between and语句

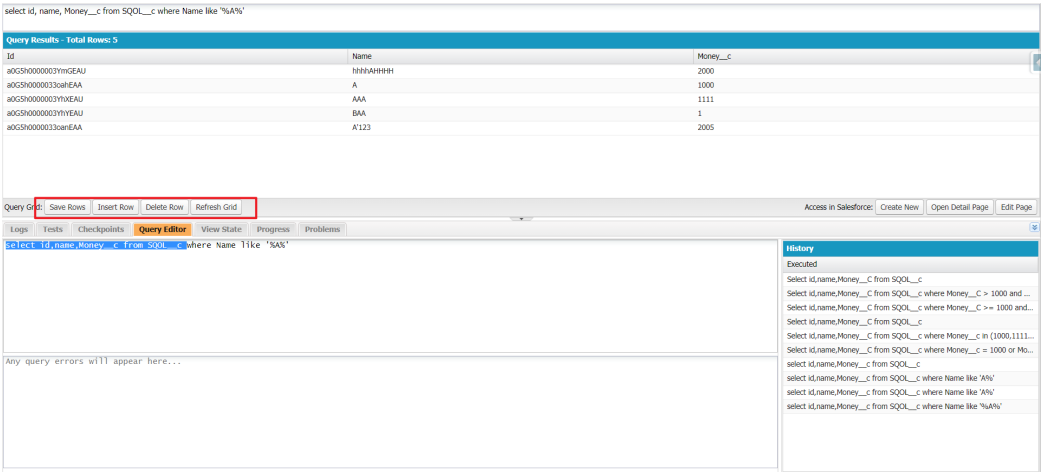
3. in、not in,or,and语句 常用

```
select id,name,Money__c from SQOL__c where Money__c in (1000,3000)
select id,name,Money__c from SQOL__c where Money__c = 1000 or Money__c = 3000
```

4. like语句

- “%”为通配符，代表0 - n个任意字符
- “-”**下划线**代表一个任意字符（有的编程语言是问号）

```
select id,name,Money__c from SQOL__c where name like 'A__'
```



5. = null ・ != null

```
select id,name,Money__c,AObject__c  from SQOL__c where AObject__c = null
```

- 不是is null
- 一个等号（= null）

6.转义字符\

```
select id,name from sqol__c where name = 'A\'123'
```

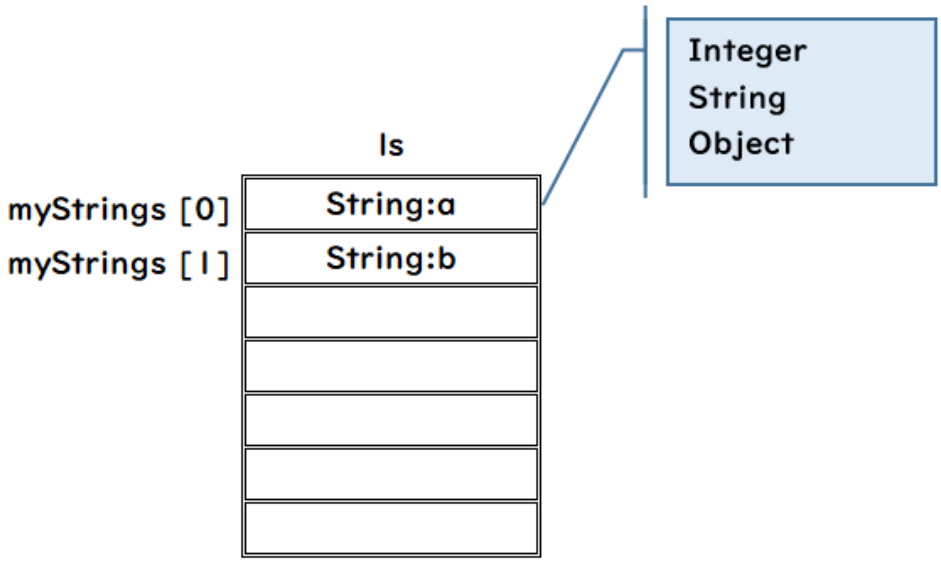
添加上转义字符（\）之后，在SOQL语句中'A\'123' 就相当于A'123

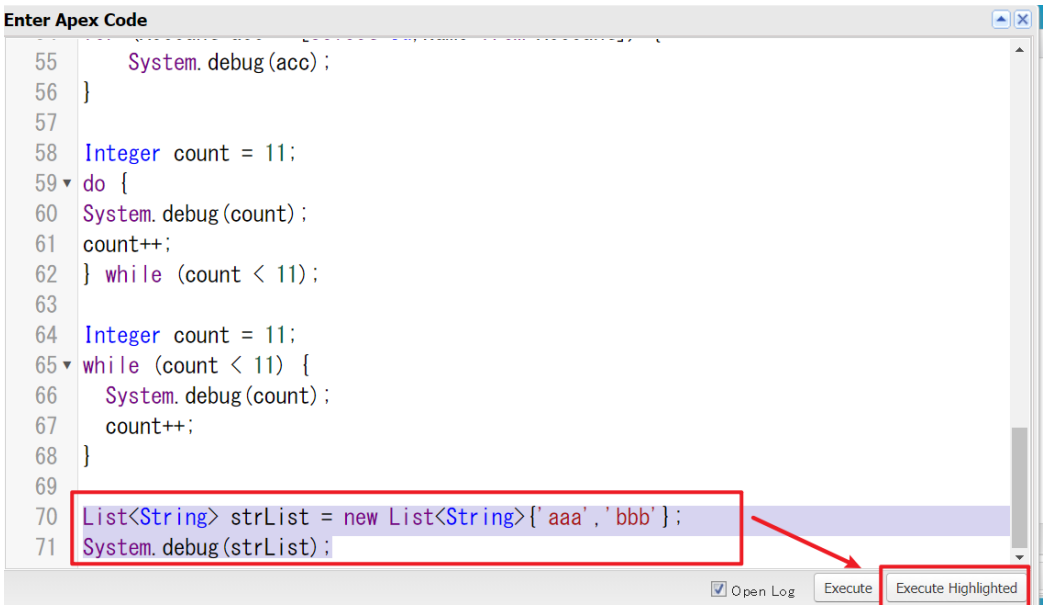
在线Salesforce编辑器

https://my-aside.herokuapp.com/

~~直接创建并完成初始化~~

```
List<String> myStrings = new List<String>{'a', 'b'};
System.debug(myStrings);
```





- [x] ~~由其他List创建~~

```
List<Integer> ls1 = new List<Integer>();
ls1.add(1);
ls1.add(2);
// Create a list based on an existing one
List<Integer> ls2 = new List<Integer>(ls1);
// ls2 elements are copied from ls1
System.debug(ls2);

List<String> strList = new List<String>{'aaa','bbb'};
List<String> ls2 = new List<String>(strList);
System.debug(ls2);

```

- [ ] 通过Set创建

```
Set<Integer> s1 = new Set<Integer>();
s1.add(1);
s1.add(2);
// Create a list based on a set
List<Integer> ls = new List<Integer>(s1);
// ls elements are copied from s1
System.debug(ls);
```

- [ ] 通过数组创建

```
List<String> colors = new String[3];
colors[0] = 'Red';
colors[1] = 'Blue';
colors[2] = 'Green';
system.debug(colors);
```

☐ 添加List中的元素(**add**)

☐ 修改List中的元素

```
List<Integer> ls = new List<Integer>();
ls.add(1);
ls.add(2);
System.debug('ls:' + ls);
ls.set(0, 3);
System.debug('ls:' + ls);
```

☐ 删除List中的元素

```
List<String> colors = new String[3];
colors[0] = 'Red';
colors[1] = 'Blue';
colors[2] = 'Green';
system.debug(colors);
String s1 = colors.remove(1);
system.debug(colors);
```

· 其他方法

```
clear();
clone();
contains(listElement);
deepClone()
isEmpty()
size()
sort()
```

8.2.2 Set

Set Class

和List类似，唯一区别是里面的数据不能重复，自动去重复元素（List中元素可重复）

Set无序

Set中没有发现修改方法

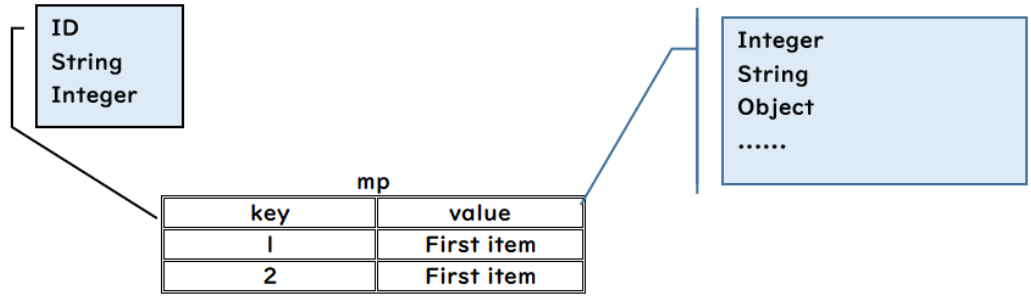
https://keneloper.com/explain-how-to-use-apex-list-map-set/

```
List<Integer> ls = new List<Integer>();
ls.add(1);
ls.add(1);
System.debug('ls:' + ls);


Set<Integer> st = new Set<Integer>();
st.add(1);
st.add(1);
System.debug('st:' + st);
```

实际项目用的多，因为可以去重复

8.2.3 Map

Map Class



- 三种创建方法

☐ 直接创建

```
Map<Integer, String> mp = new Map<Integer, String>();
mp.put(1, 'First item');
mp.put(2, 'Second item');
System.debug(mp);

{1=First item, 2=Second item}

Map<Integer, String> mp = new Map<Integer, String>();
mp.put(1, 'First item');
mp.put(1, 'Second item');  //覆盖掉key相同的元素
System.debug(mp);
{1=Second item}
```

☐ 通过其他map创建

```
Map<Integer, String> m1 = new Map<Integer, String>();
m1.put(1, 'First item');
m1.put(2, 'Second item');

Map<Integer, String> m2 = new Map<Integer, String>(m1);
System.debug(m2);

{1=First item, 2=Second item}
```

☐ 通过List创建

```
List<Account> ls = [select Id,Name from Account limit 1];
Map<Id, Account> mp = new Map<Id, Account>(ls);
System.debug(mp);

{0012r0000078jkJAAQ=Account:{Id=0012r0000078jkJAAQ, Name=Edge Communications
```

☐ 取得Key方法（keySet）

```
Map<String, String> colorCodes = new Map<String, String>();

colorCodes.put('Red', 'FF0000');
colorCodes.put('Blue', '0000A0');

Set <String> colorSet = new Set<String>();
colorSet = colorCodes.keySet();
System.debug(colorSet);

：{Blue, Red} //Map中是无序的
```

☐ 取得值方法（get）

```
Map<String, String> colorCodes = new Map<String, String>();

colorCodes.put('Red', 'FF0000');
colorCodes.put('Blue', '0000A0');

System.debug(colorCodes.get('Blue'));

0000A0
```

☐ 添加方法（put, putAll）

- **put**
- **putAll(fromMap)**

```
Map<String, String> map1 = new Map<String, String>();
map1.put('Red','FF0000');
Map<String, String> map2 = new Map<String, String>();
map2.put('Blue','0000FF');
// Add map1 entries to map2
map2.putAll(map1);
System.assertEquals(2, map2.size());
```

- **putAll(sobjectArray)**

```
List<Account> accts = new List<Account>();
accts.add(new Account(Name='Account1'));
accts.add(new Account(Name='Account2'));
// Insert accounts so their IDs are populated.
// 如果把下面这句话屏蔽掉，可以吗？
insert accts;
Map<Id, Account> m = new Map<Id, Account>();
// Add all the records to the map.
m.putAll(accts);
System.assertEquals(2, m.size());
```

```
Map<Integer, String> m1 = new Map<Integer, String>();
m1.put(1, 'First item');
m1.put(2, 'Second item');

Map<Integer, String> m2 = new Map<Integer, String>(m1);

m1.put(1,'third');

System.debug(m1);
System.debug(m2);
////////////////////////////////////
Map<Integer, String> m1 = new Map<Integer, String>();
m1.put(1, 'First item');
m1.put(2, 'Second item');

Map<Integer, String> m2 = m1;

m1.put(1,'third');

System.debug(m1);
System.debug(m2);
//////////////////////////////

List<Integer> ls1 = new List<Integer>();
ls1.add(1);
ls1.add(2);
// Create a list based on an existing one
List<Integer> ls2 = new List<Integer>(ls1);

ls1.set(0,3);
// ls2 elements are copied from ls1
System.debug(ls1);
System.debug(ls2);


List<Integer> ls1 = new List<Integer>();
ls1.add(1);
ls1.add(2);
// Create a list based on an existing one
List<Integer> ls2 = ls1;

ls1.set(0,3);
// ls2 elements are copied from ls1
System.debug(ls1);
System.debug(ls2);
```

☐ 删除方法（remove）

```
Map<String, String> colorCodes = new Map<String, String>();

colorCodes.put('Red', 'FF0000');
```

```
colorCodes.put('Blue', '0000A0');

String myColor = colorCodes.remove('Blue');
String code2 = colorCodes.get('Blue');

System.assertEquals(null, code2);
```

```
Map<Integer, String> m1 = new Map<Integer, String>();
m1.put(1, 'First item');
m1.put(2, 'Second item');

Map<Integer, String> m2 = m1;

m1.put(1,'third');

System.debug(m1);
System.debug(m2);


List<Integer> ls1 = new List<Integer>();
ls1.add(1);
ls1.add(2);
// Create a list based on an existing one
List<Integer> ls2 = new List<Integer>(ls1);

ls1.set(0,3);
// ls2 elements are copied from ls1
System.debug(ls1);
System.debug(ls2);
```

8.2.4 clone()与deepClone()

・对于基本类型(<>中是基本类型)，无法使用deepClone()，只能使用clone

・对于SObject 类型，deepClone可以复制出一个完全没有联系的副本

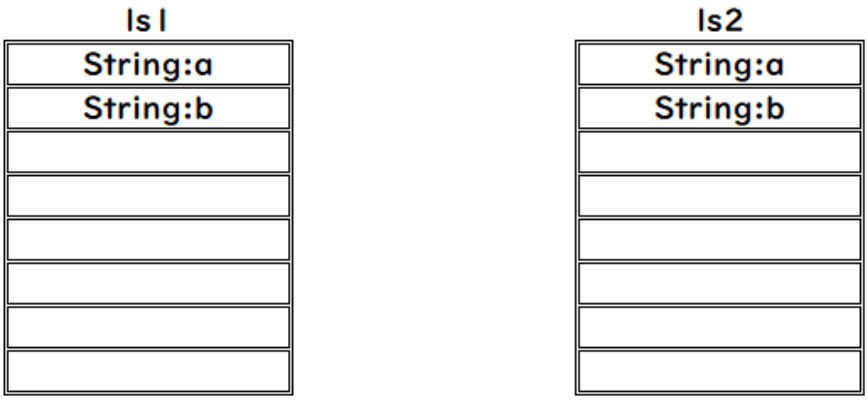
※如果要使用深度克隆，有需要的话，要创建一个自定义类

基本类型：

```
List<String> ls1 = new List<String>();
ls1.add('a');
ls1.add('b');
System.debug(ls1);

List<String> ls2 = ls1.clone();
System.debug(ls2);

ls1.set(0,'c'); // 修改ls1中的第一个元素

System.debug('modify after:' + ls1);
```

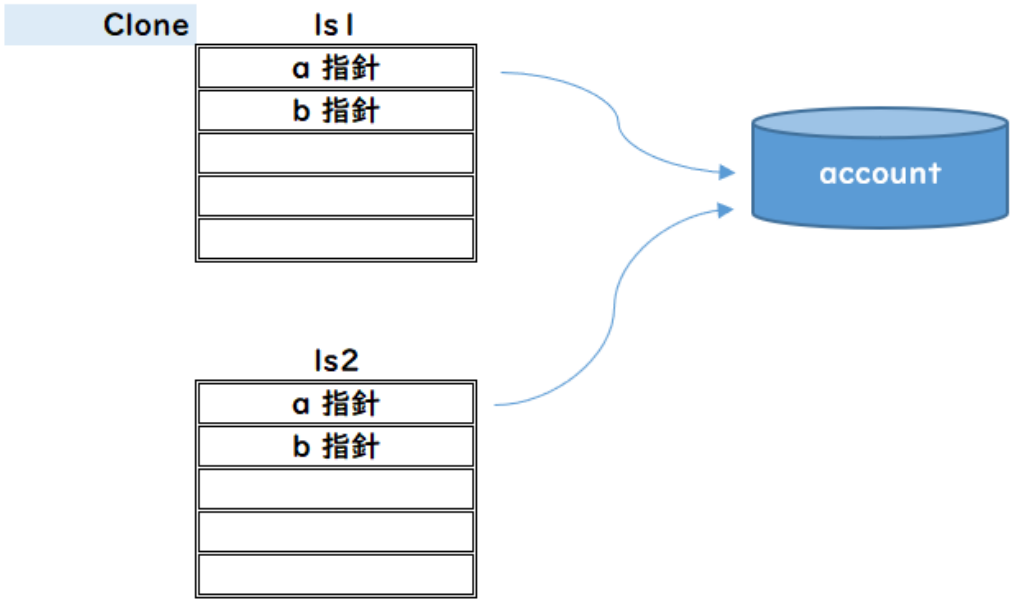
```
System.debug('modify after:' + ls2);
```

```
List<String> ls2 = ls1.deepclone();

当使用deepClone的时候出现如下错误：
Operation only applies to SObject list types: List<String>
```

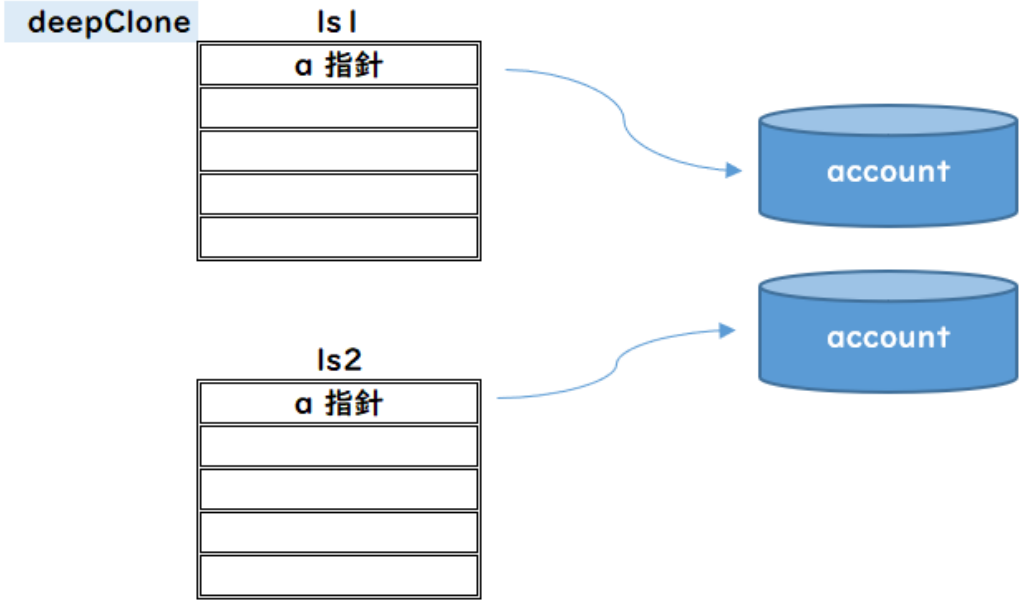


引用类型：

```
List<Account> ls1 = new List<Account>();

Account a = new Account(Name='Acme', BillingCity='New York');
ls1.add(a);

List<Account> ls2 = ls1.clone();
System.debug(ls1);
System.debug(ls2);

Account a1 = ls1.get(0);
a1.BillingCity = 'San Francisco';

System.debug('modify after ls1 :' + ls1);
System.debug('modify after ls2 :' + ls2);

```




```
List<Account> ls1 = new List<Account>();

Account a = new Account(Name='Acme', BillingCity='New York');
ls1.add(a);

List<Account> ls2 = ls1.deepClone();
System.debug(ls1);
System.debug(ls2);

Account a1 = ls1.get(0);
a1.BillingCity = 'San Francisco';

System.debug('modify after ls1 :' + ls1);
System.debug('modify after ls2 :' + ls2);
```



List<**String/Integer 11中基本类型**>的话只能用clone，而且只是分开保存的

List<**Account/自定义的都现象**>的话，可以用clone和deepclone，

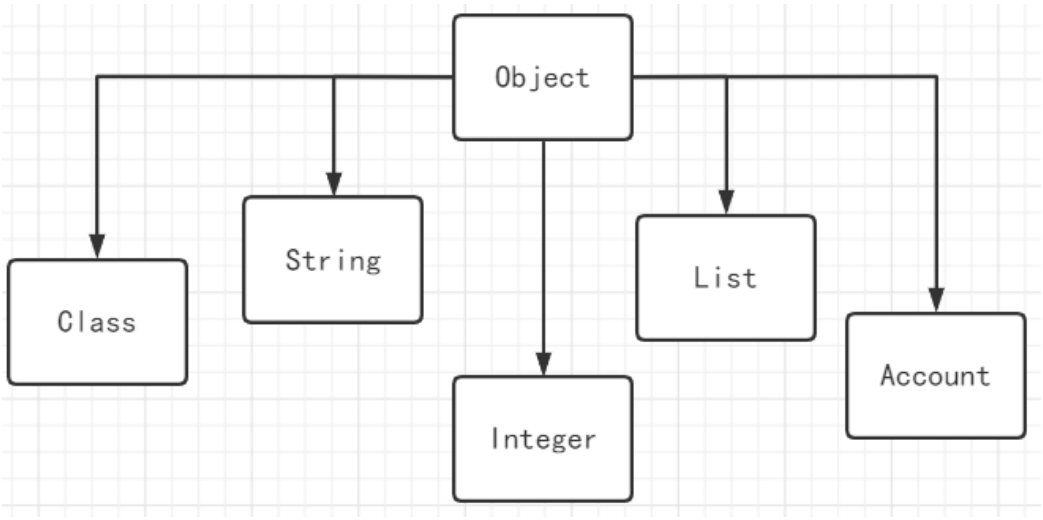
第一种是同一个对象，ls1和ls2指向相同，

第二种完全是相当于两个不同的对象

8.3 Object类型

SObject Class

※Object类型是一切类型的父类




```
Ex.1
Object obj = 10;
Integer i = Integer.valueof(obj);
System.debug(i);

Ex.2
Object obj = 'Ex.2';
String s = (String)obj;
System.debug(s);

Ex.3
// 不要和主从关系搞混了，不是一回事
List<Object> accList = [SELECT Id, Name,MasterRecordId FROM Account];

if (accList[0] instanceof Account) {
    System.debug('this is a Account instance!');
}

```

8.4 枚举类型

・Enum

枚举类型包括三个方法

1. values

2. name
3. ordinal

```
public enum Season {WINTER, SPRING, SUMMER, FALL}

List<Season> values = Season.values();
Integer i = Season.SUMMER.ordinal();
String s = Season.SUMMER.name();

// i:2
System.debug('i:' + i);

// s:SUMMER
System.debug('s:' + s);

// values:(WINTER, SPRING, SUMMER, FALL)
System.debug('values:' + values);

Season ss = Season.SPRING;
Season m(Integer x, Season o){
   if(ss ==Season.SUMMER) return o;
     //….
}
```

说明文档

- System.StatusCode
- System.XmlTag
- System.LoggingLevel
- System.RoundingMode
- System.SoapType
- System.DisplayType
- System.JSONToken
- ApexPages.Severity
- Dom.XmlNodeType

```
 List values = StatusCode.values();
 System.debug(values);

 String s = StatusCode.MISSING_ARGUMENT.name();
 System.debug(s);

Integer i = StatusCode.MISSING_ARGUMENT.ordinal();
System.debug(i);

//
UNKNOWN_EXCEPTION, TOO_MANY_APEX_REQUESTS, MISSING_ARGUMENT, INVALID_ARGUMEN
```

8.4.1 再说System.debug()

System.debug()是可以加第一个参数的

Logging Level

```
System.LoggingLevel level = LoggingLevel.ERROR;

System.debug(logginglevel.INFO, 'MsgTxt');
```

第9章 定数（常量）

定数就理解成固定的数，就是不变的数,赋值之后，无法改变

使用static final关键字,static可以不用,

但是，但是一般都是两个关键字一起用

通常全部大写

- **static final定义**
- **赋值之后，无法改变**

```
static final Integer PRIVATE_INT_CONST = 200;
static final Integer PRIVATE_INT_CONST2;

static final String PRIVATE_STR_CONST = 'AAAA';

PRIVATE_INT_CONST2 = 300;
// PRIVATE_INT_CONST2 = 400;

System.debug('PRIVATE_INT_CONST2:' + PRIVATE_INT_CONST2);
System.debug('PRIVATE_INT_CONST :' + PRIVATE_INT_CONST2);

//PRIVATE_INT_CONST2 = 400;
//System.debug('PRIVATE_INT_CONST2:' + PRIVATE_INT_CONST2);
```

第10章 运算符(演算子：えんざんし)

说明文档

10.0 基本演算子

```
1.Ex
Integer x = 10;
Integer y = 20;

```

```
x = y;
System.debug(x);

2.Ex
x += y;
x = x + y; //30

3.Ex 自增自减
i++;i--;  先赋值，后自增
++i;--i;  先自增，后赋值

Integer base = 3;
Integer m = base++;
System.debug('m=' + m);
System.debug('base=' + base);
Integer n = ++base;
System.debug('n=' + n);
System.debug('base=' + base);


4.Ex（移位运算）
x >>= y

Integer x = 10;
System.debug(x >>= 2);

10的2进制是1010 向右移动1位 0101
0000      0
0001      1
0010      2
0011      3
0100      4
0101      5

0，1，2，3，4，5，6，7 10
0，1，2，3，10
0，1，2，3，4，5，6，10


冯·诺依曼
设计思想之一是二进制，他根据电子元件双稳工作的特点，建议在电子计算机中采用二进制。
报告提到了二进制的优点，并预言，二进制的采用将大大简化机器的逻辑线路。

什么是2进制，就是用0和1对应电子设备的开和关的两种状态（只有两种）
比如说上电（通电）就是1，下电（不通电）就是0

计算机的世界都是【0，1，0，1】这样的数字构成，因为只有0和1两个字，所以叫2进制
也对应了电子设备的开和关的两种状态

现实世界是10进制，所以有【0，1，2，3，4，5，6，7，8，9】这10个数
实际上是产生了进位，变成了一零
```

不过，几千年前，易经中就用阴阳来划分世界万物，不知道冯·诺依曼是不是借鉴了易经的思想

10.1 按位与

「&」、「|」

- 可以用作位运算符

按位与规则：有零为零，相当与乘法

例子：12 & 5 = 4

```
System.debug(12 & 5);  // 结论是：4
```

1	1	0	0		
0	1	0	1		
0	1	0	0		0100=4

- 也可以用逻辑运算符号

```
例子：true & false = falseSystem.debug(12 & 5);  // 4
System.debug(true & false); OK

System.debug(true & 10); NG
```

结论：当你不涉及到位运算的时候，尽量不要使用一个的。

(&&, ||)

10.2 按位或

```
System.debug(12 | 5); // 结论是13
```

1	1	0	0		
0	1	0	1		
1	1	0	1		
第3位	第2位	第1位	第0位		
1*(2^3)	1*(2^2)	0*(2^1)	1*(2^0)		8+4+0+1=13
	底都是2				

```
System.debug(true | false);
```

```
System.debug(false| false);
System.debug(true| true);
System.debug(false| true);
```

结论：当你不涉及到位运算的时候，尽量不要使用。

10.3 && 短路与

第一个如果是false，后面的语句不执行，结果为false

- 只要有一个为false，结果为false

```
Integer x = 10;
Integer y = 20;
Integer z = 30;
Integer m = 40;

// System.debug(只能是布尔类型);
System.debug(x > y);
System.debug(x > y && true);
System.debug(x > y && false);
```

10.4 || 短路或

第一个如果是true，后面的语句不执行，结果为true

- 只要有一个为True，结果为True

```
Integer x = 10;
Integer y = 20;
Integer z = 30;
Integer m = 40;

// System.debug(只能是布尔类型);
System.debug(x > y);
System.debug(x > y || true);
System.debug(x > y || false);
```

10.5 x == y

```
String x = '10';
String y = '20';

System.debug(x == y);

List<Integer> ls = new List<Integer>();
ls.add(1);

List<Integer> ls1 = new List<Integer>();
ls1.add(1);

```

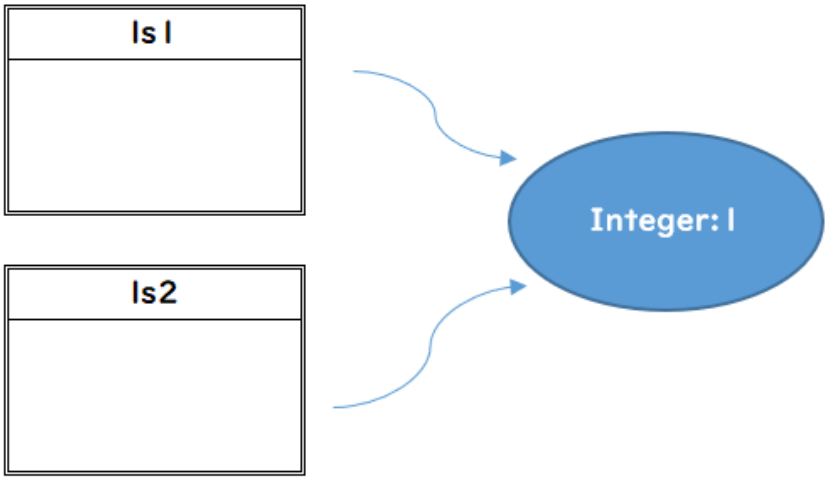
```
System.debug(ls == ls1);
```

10.6 x === y

这个操作符只能用于**引用类型**

比较严格，不仅要值相等，而且要内存中的位置也相等

```
List<Integer> ls = new List<Integer>();
ls.add(1);

List<Integer> ls1 = new List<Integer>();
ls1.add(1);

System.debug(ls == ls1);
System.debug(ls === ls1);

List<Integer> ls2 = ls1;

System.debug(ls1 == ls2);
System.debug(ls1 === ls2);

```



第11章 类型转换

11.1 自动类型转换

从小到大可以自动类型转换，反之则不行

```
Integer i = 10;
Long j = i;
Double k = j;
Decimal l = k;

System.debug('j:' + j);
System.debug('k:' + k);
System.debug('l:' + l);
```

从小到大可以隐式转换，或者说自动转换，颠倒就报错

类型转换的例子：

```
String a = '1001';
Integer aa = Integer.valueOf(a) + 1;
System.debug(aa);
String b = 'AA-' + String.valueOf(aa);
System.debug(b);
```

11.2 强制类型转换

最常用的一个函数 类型转换

1. 类型.valueof(值) 个人建议用这种

2. (类型)值

```
String accId = String.valueof(accList[0].Id);
String accId1 = (String)accList[0].Id;
```

```
Integer i = 10;
String str1 = String.valueof(i);
System.debug(str1);

String str2 = '100';
Integer j = Integer.valueof(str2);
System.debug(j)

Decimal a = 10.3;
Integer b = Integer.valueOf(a);
System.debug('b:' + b);
```

11.3 ID与String

- ID转换成String是可以的

```
List<Account> accList = [SELECT Id, Name,MasterRecordId FROM Account];
String accId = String.valueof(accList[0].Id);
String accId1 = (String)accList[0].Id;
System.debug(accId1);

```

- String转换成ID是有危险的

```
String str = '0015h00000Bds32AAB';
Id id1 = Id.valueOf(stjava);
String str = 'aaa132';
Id id1 = Id.valueOf(str);
System.debug(id1);
```

- 需要转换之前用instanceof 来判断

```
// String str = '0015h00000Bds32AAB';
String str = 'aaa123';

if (str instanceof ID) {
    Id id1 = Id.valueOf(str);
    System.debug(id1);
} else {
    System.debug('str is not a id!');
}
```

第12章 控制语句（if, else, switch）

12.1 if else

```
// 中文就是如果，否则
// 中文编程易语言和VB6.0很类似 做外挂
```

```
如果 ([Boolean_condition])    // 如果
  // Statement 1
否则                          // 否则
  // Statement 2

Integer score = 95;

if(score > 90) {
  System.debug('有奖励');
} else {
  System.debug('挨揍');
}
```

```
Integer x, sign;
// Your code
if (x <= 0) {
  if (x == 0) {
    sign = 0;
  } else {
    sign = -1;
  }
}
```

```
if (place == 1) {
  medal_color = 'gold';
} else if (place == 2) {
  medal_color = 'silver';
} else if (place == 3) {
  medal_color = 'bronze';
} else {
```

```
  medal_color = null;
}
```

```
Integer score = 100;

if(score > 90) {
    if(score > 95){
        System.debug('游乐场一天');
    }else {
        System.debug('买个小玩具！');
    }
} else {
  System.debug('挨揍');
}
```

12.2 三目运算符

x ? y : z

相当于：if-then-else，而且x不能是空

```
Integer score = 100;

String result = score > 90 ? 'OK':'NG';
System.debug(result);

```

12.3 switch

12.3.1 Ex1

```
switch on expression {
when value1 { // when block 1
// code block 1
}
when value2 { // when block 2
// code block 2
}
when value3 { // when block 3
// code block 3
}
when else { // default block, optional
// code block 4
}
}
```

12.3.2 Ex2

```
// Integer i; 不要给i赋值，让i是null
Integer i = 2;
switch on i {
when 2 {
System.debug('when block 2');
}
when -3 {
System.debug('when block -3');
}
when null {
  System.debug('bad integer');
}
when else {
System.debug('default');
}
}

// if 改写
Integer i = 2;
if (i == 2) {
    System.debug('when block 2');
} else if (i == -3) {
    System.debug('when block -3');
} else if (i == null) {
    System.debug('bad integer');
} else {
    System.debug('default');
}
```

12.3.3 Ex3

```
switch on i {
// if (i == 2 or i ==3 or i == 4)
when 2, 3, 4 {
System.debug('when block 2 and 3 and 4');
}
when 5, 6 {
System.debug('when block 5 and 6');
}
when 7 {
System.debug('when block 7');
}
when else {
System.debug('default');
}
}
```

12.3.4 Ex4

```
String str = 'a';
switch on str {
when 'a', 'b', 'ccc' {
System.debug('when block 2 and 3 and 4');
}
when else {
System.debug('default');
}
}
```

```
List<SObject> objList = [Select id,name from Account limit 1];

switch on objList[0] {
    when Account a {
        System.debug('account ' + a);
    }
    when Contact c {
        System.debug('contact ' + c);
    }
    when null {
        System.debug('null');
    }
    when else {
        System.debug('default');
    }
}
```

第13章 循环语句（for, while...）

- ☐ do {statement} while (Boolean_condition);
- ☐ while (Boolean_condition) statement;
- ☐ for (initialization; Boolean_exit_condition; increment) statement;
- ☐ for (variable : array_or_set) statement;
- ☐ for (variable : [inline_soql_query]) statement;

13.1 do while

do {操作内容} while(退出条件);

```
Integer count = 1;
do {
System.debug(count);
count++;
} while (count < 11);
```

分析:

> Step1. 先执行一次do中的语句
> Step1. 判断count是否小于11
> Step2. 小于11，继续执行do中的语句，一定要有count++语句
> Step3. 判断count是否小于11
> Step4. 小于11，继续执行do中的语句，

13.2 while

while（退出条件）{操作内容}

```
Integer count = 1;
while (count < 11) {
  System.debug(count);
  count++;
}
```

分析：

> Step1. 判断count是否小于11
> Step2. 小于11，执行do中的语句，一定要有count++语句
> Step3. 判断count是否小于11
> Step4. 小于11，执行do中的语句，count为2

do {操作内容} while(退出条件);
while（退出条件）{操作内容};
※不同之处在于，do while先执行一次，把上述代码改成11，在分别执行一次

```
Integer count = 11;
do {
System.debug(count);
count++;
} while (count < 11);

Integer count = 11;
while (count < 11) {
  System.debug(count);
  count++;
}
```

13.3 for

13.3.1 Ex1

```
for (init_stmt; exit_condition; increment_stmt) {
```

```
  code_block
}

for (Integer i = 0; i < 10; i++) {
  System.debug(i);
}
```

分析：

> Step1. Integer i = 0; 程序开始执行（初始化操作，所以只执行一次）
> Step2. 判断i是否小于10，这个时候的i为0，小于10
> Step3. 小于10，执行System.debug()语句
> Step4. 然后执行i++语句，i变为1
> Step5. 判断i是否小于10，这个时候的i为1，小于10
> Step6，小于10，执行System.debug()语句
> Step7. 然后执行i++语句，i变为2

continue和break关键字

```
// 实际项目中，当循环查找某个值的之后，找到之后，就直接Break
for (Integer i = 0; i < 10; i++) {
    if (i == 8) {
        break;
    }
    System.debug(i);
}

for (Integer i = 0; i < 10; i++) {
    if (i == 8) {
        continue;
    }
    System.debug(i);
}
```

13.3.2 Ex2

```
for (variable : list_or_set) {
  code_block
}

List<String> strList = new List<String>{'aa','bb'};

for (String str : strList) {
    System.debug(str);
}
```

13.3.3 Ex3

```
for (variable : [soql_query]) {
  code_block
}

for (Account acc : [Select Id,Name from Account]) {
    System.debug(acc);
}
```