Value Representation, Hashing, and Generics

2019年12月17日 21:24

Low-Level Representation

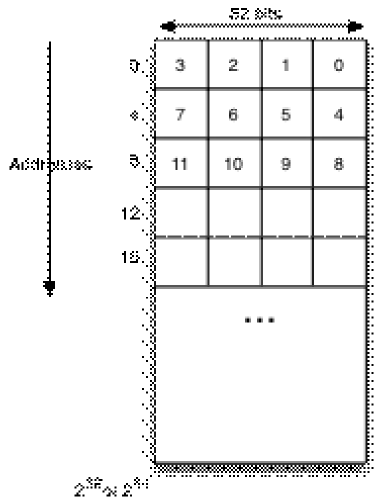
The memory can be accessed using memory addresses that start at 0 and go up to some large number (usually 2^32 or 2^64, depending on whether the computer is 32-bit or 64-bit).

注意 double word & word 的定义是可变的，有的时候 1 word = 2 bytes，那可能是很老的电脑，但无论如何 word 都是电脑再将 memory addresses 抽象化之后一行的那个基础单位

1 double word = 2 words	1 word = 4 bytes	1 byte = 8bits

	<-8bits->
0	
1	
2	
…	

You might think computer stores memory in this way, but in fact it offers a more abstract way to view it as seeing it as a 2D array.



No matter whether a 32-bit or 64-bit machine this is, the columns will always be 32 bits, the difference is in the number of rows.

byte	1 byte
short	2
char	2
int	4
long	8

Signed Representation

In one byte,

7	6	5	4	3	2	1	0
the most significant bit (the end of this byte)							the least significant bit (everything starts from here)

2's compliment representation:

In representing a signed value, such as a byte or an int, the leftmost(most significant) bit is a sign bit, which stands for -2^(n-1) instead of 2^(n-1)
e.g.

13

0	0	0	0	1	1	0	1

127

0	1	1	1	1	1	1	1

-1

1	1	1	1	1	1	1	1

sign bit
最高位(sign bit) =1 时，我们有 -128
所以-1+1 =0，因为从 least significant digit +1 一直进位，直到最后一个 进出来的1 超出了byte的表示范围，于是就=0了

相当于本来 8bits 可以代表 0~255，现在我们把 128~255 那一部分切下来放到0的右边，相当于这一部分代表的数-256，现在代表 -128~-1 ，由此一来，我们得到8bits代表 -128~127

	0~127	128~255	unsigned
-128~-1	0~127		signed

Bit Operation

negation

negation is just take all the bits and flip them over
There's an interesting property of negation:
~x negation = -1-x
~x+1 = -x
只看 ~x 不太容易看出来为什么是 -1-x ， 但是如果从性质 ~x+x+1=0 开始推就很简单能看出来

a	1	1	0	0	0	0	1	0

$\neg a$	0	0	1	1	1	1	0	1
$a + \neg a$	1	1	1	1	1	1	1	1

那么我们再加一个1，就是0了，这在二进制中是很显然的，因为进位进出来的1跑到第9位去了，所以现在我们有的8位都是0，十进制中也很显然，只是要记住 the most significant bit 是负数，那么+1以后 前7位就和第八位抵消了

exclusive or

仅当不同才会给1：

1 ^ 1 = 0
0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 0

shift

- arithmetic shift:

  high bit stays

1	0	1	0
1	1	0	1

  1010>>1，得到 1101

  101右移一位，high bit stays 1

  实际上是得到了我们预想的 *2 | /2 的结果
- logic shift:

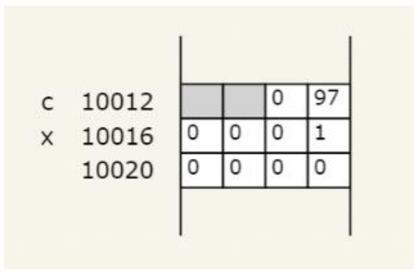
  1010 整体右移一位，空出来的位置用0补全

  实际上在算术上没有意义，但是如果你用二进制存储东西，这是有意义的

Variables and Arrays

Variables:

variables are stored in memory on word boundaries and always take up an integral number of words.So variables with types short, int, byte, and char all take up one full 32-bit word, whereas variables with type long take up two consecutive words.

char c = 'a';
long x = 1;
	 c
	10012
	0
	97
	x
	10016
	0
	0
	0
	1
	10020
	0
	0
	0
	0

Arrays:

Unlike variables, array elements are usually stored in a more compact way in memory in which no space is wasted. An array of characters (a char[] value) is stored with the characters packed contiguously in memory, with two characters per 32-bit word. Similarly, a byte[] array is stored with each byte of the array taking up exactly one byte in memory.

The exception is arrays of boolean, which are stored with one full word per boolean value. It is therefore prudent to avoid the type boolean[] for large arrays.

char[] s = {'h','l','b'}

char[]						
(length =) 3						
0	105	0	104			
-------	-------	0	98			
byte[]						
(length =) 2						
-------	-------	42	40			

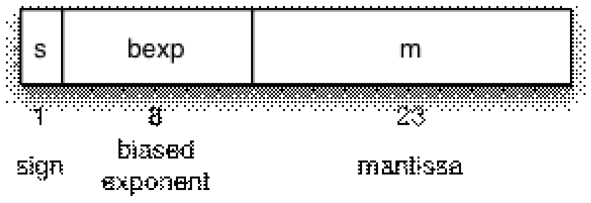
Pointers

In a 64 bit machine

指针（变量名）存储一个 2-words-long (double word) 的地址

In a 32 bit machine, it stores a 1-word-long address.

Floating & Double



`(−1)^s·2^exp·(1.m)`

***m:** Here, we interpret m as a sequence of binary digits, so "1.m" represents a binary number that is at least 1 (and equal to 1 in the case where m = "000000...") and less than 2. The maximum number that 1.m can represent is "1.11111...", which is a binary representation of a number less than 2 by the small amount 2−23.*

***exp:** the exponent is between −126 and 127*

e.g.

Thus, if we want to represent the number 1, we choose s=0, exp=0, m=0000... . To represent numbers outside the interval [1,2), an exponent is needed. To represent 2, we use s = 0, exp = 1, m=0000..., since $1\cdot 2^1$ = 2. To represent 3, we choose s=0, exp=1, m=10000..., since 1.1 in binary represents 3/2, and $3/2\cdot 2^1$ = 3. And so on.

Hashing Table

- chaining: 每个 bucket 实际上是一个 动态数组/链表
- probing: 当发生冲突，就向下找，然后放到最近的一个空格子里

Load factor

<u>When we search for an element that is not in the hash table, the expected length of the linked list traversed is α.</u>

$\alpha = \frac{n}{m}$

(a: load factor; n: elements in the hashTable; m: buckets in the hashTable)

- chaining: $\frac{1}{2} < \alpha < 1$
    - >1: probably there are many collisions. The cost of traversing the list can limit performance, therefore every action can't be $O(1)$
    - < 1/2: the array is sparse, not fully utilized
- probing:
  for probing, you want a to be even smaller, because if a is 1, when there is a collision, you always have to go through the whole linked list
  Therefore, the loading factor of probing is always smaller than that of chaining

HashTable should resize according to this load factor

Amortized Complexity

starting a hashing table of only 1 bucket, add 2^j elements, rehashing on all powers of 2

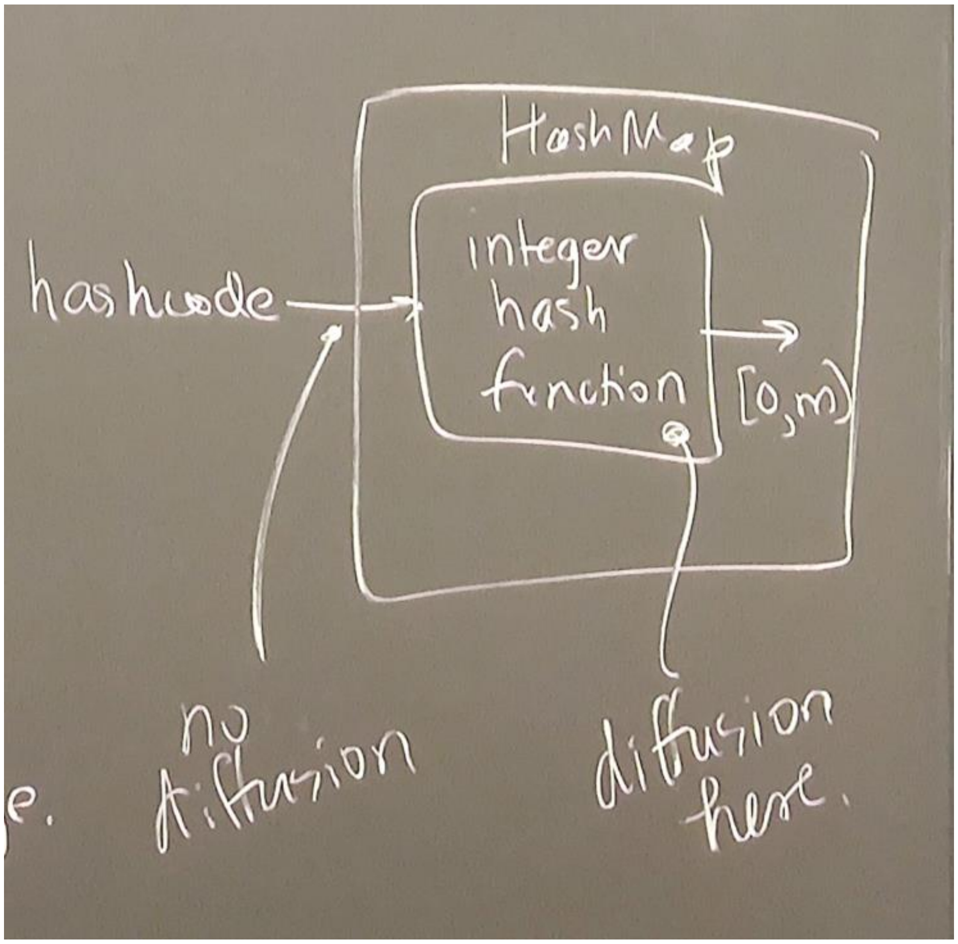
initial hashing	refreshing
2^j	1+2+4+ ... + 2^j = 2^(j+1)

$sum = 3 * 2^j$，所以单个操作平均下来是 $\frac{3 * 2^j}{2^j} = 3$

我们本来说 add是 O(1) 的，现在我们看到最坏情况也就是 O(3) 的

Starting with a table of size 1, say we add a sequence of n = 2j elements. The hash table resizes after 1 add, then again after 2 more adds, then again after 4 more adds, etc. Not counting the array resizing, the cost of adding the n elements is O(n) on average. The cost of all the resizings is (a constant multiple of) 1 + 2 + 4 + 8 + ⋯ + 2j = 2j+1 − 1 = 2n − 1, which is O(n).

Hash 的最终状态：将n个元素均匀分布到 $key \subset [0, m)$中



Hash Code

<u>any integer value reliably (injective) points the Object to an integer (however, doesn't provide diffusion)</u>

- hashCode must agree with equals method
  if you have two Strings have the same contents, they should return the same hash value
- should be injective
  - use all range of int
  - use all data in key: change to key, change hashCode

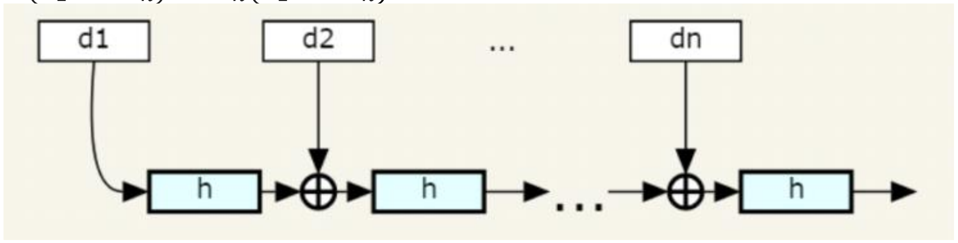
- Mutable Objects: we can use their addresses as the hashCode - satisfy both
- Immutable Objects: if we use memory address as hashCode, - satisfy injective, but doesn't satisfy "equals"

Hash Code of Immutable Data Structures

Immutable data abstractions such as String would normally define their notion of equality in terms of the data they contain rather than their memory address. The hashCode() method should therefore return the same hashCode for objects that represent the same data value.

a good hash function H can be constructed iteratively by feeding the output of H on all previous values to h

$H_1(d_1) = h(d_1)$
$H_{i+1}(d_1,\ldots, d_{i+1}) = h(H_i(d_1,\ldots, d_i) \oplus d_{i+1})$
$H(d_1, \ldots, d_n) = H_n(d_1, \ldots, d_n)$



Note that this is very different from the alternative of hashing all the individual data values and combining them with exclusive-or. That perhaps tempting algorithm would tend to have more collisions because the commutativity and associativity of exclusive-or would cause that algorithm to give the same result on any permutation of the data values.

If writing a loop to implement the diagram above seems like too much work, a useful trick for constructing a good hashCode() method is to leverage the fact that Java String objects implement hashCode() in this way, using the characters of the string as the data values. If your class defines a toString() method in such a way that it returns the same strings exactly when two objects are equal, then hashCode() can simply return the hash of that string!

```
  // Requires: o1.toString().equals(o2.toString()) if and only if
o1.equals(o2)
  int hashCode() {
    return toString().hashCode();
  }
```

One Java pitfall to watch out for arises because Java's collection classes also override hashCode() to compute the hash from the current contents of the collection, as if Java collections were immutable. This way of computing the hash code is dangerous, precisely because collections are not immutable. Mutating the collection used as the key will change its hash code, breaking the class invariant of the hash table. Any collection being used as a key must not be mutated.

其中⊕是异或的意思
取异或的意义：将两个数据（前面所有数据的hash 和 这个新进来的数据）整合起来

Hash Function

provides "diffusion"/ "randomness" and diffuses it to a random

- Modular Hash: choose prime number p
  $h(k) = k \% p$
- Multiplicative Hash:
  $$h(k) = \lfloor m * frac(k \cdot A) \rfloor$$
  m is the number of buckets,

A is a number between $0\sim1$,
$frac(x)$ gives the fraction part of x ,why takes the fraction part? Fraction part combines all parts of information together very well

The following reasoning assumes w (one word length) = 32

$h(k) = [(A * k)\% 2^w] \gg (w - r)$

a. A is a w-bit integer, which is obtained by multiplying the A in the previous equation by 2^w. $A(32 - bit\ length) = A(\subset \{0\sim1\}) * 2^{32}$
b. k should also be a w bit integer
c. $(A * k)$ binary multiplication is done by making a copy of k at eveyr bit where A has a 1, and add all those copies together, therefore, 2 32-bit numbers multiply together, the result would be a 64-bit number
d. w is the bit of the computer (w=32 for 32-bit machine)
e. $(A * k)\% 2^w$ takes the first/right (32-bit) chunk of the result
f. r is how many btis of information you would like for the results
g. $\gg w$ brings us to the end (the leftmost point) of the number
h. $\ll r$ gives us a chunk, whose length is r, starting from the end of the chunk
i. If we have a table of length $m = 2^r$, the number in that chunk will be $0\sim2^r - 1$, therefore it distributes evenly to the table
为什么要中间的这 r bit？这是因为中间的数据是整个乘数的一个混合，整合了多方面的信息

MIT 6.006 8 Hashing

在java中，因为 int 只有32位，所以 $\% 2^w$ 这一步是可以省掉的，因为后面那32位直接overflow了，被程序自己舍弃掉了

Unfortunately, multiplicative hashing is often implemented incorrectly and has unfairly acquired a bad reputation in some quarters because of it. A common mistake is to implement it as $(\mathrm{k} \cdot \mathrm{A\ mod\ m})$. *By the properties of modular arithmetic,* $(\mathrm{k} \cdot \mathrm{A})\ \mathrm{mod\ m} = ((\mathrm{k\ mod\ m}) \times (\mathrm{A\ mod\ m})\ \mathrm{mod\ m})$. *Therefore, this mistaken implementation merely shuffles the buckets rather than providing real diffusion.*

但是我们上面的那个是可以

Generics

- Subtype Polymorphism: Object Oriented
- Parametric Polymorphism: Generics

Collections in Java5

Even though you want to create a collection of integers, you can add other Objects into it. When you want to use the element in this collection, you have to downcast that element to use it. All this is because it doesn’t't' have static-type check.

Arrays:

<u>built-in parameterized type</u>
How to think of arrays? Integer[] == a function: array(Integer) outputs a type
We can think of the type Integer[] as the application of a type-level function (we might call it array if we were to give it a readable name) to the type parameter Integer and returning the type Integer[].
transforms Integer into a type that acts in the way array does
It is actually a function called "array" that takes in several objects of Integer types and transforms them to another type of Integers boxed together

so Collection is a function that takes type T and transforms it into another type
The idea of generics is to allow the programmer to define their own parameterized types, to obtain the same static checking that is available with the built-in array type.

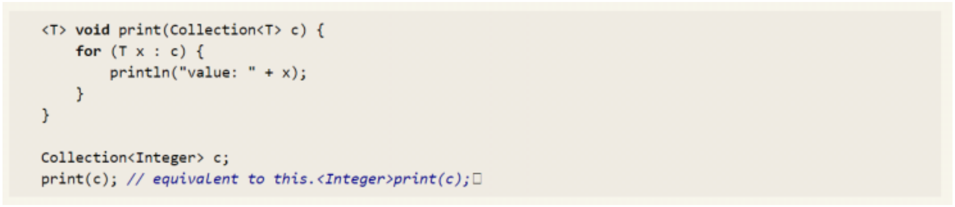
Type parametrization

Collection<String> == Collection<T> with T replaced by String

```
Collection<String> c = ...
c.add("hi"); // checked!
c.add(2); // illegal: static error
for (String s : c) {
    // use s
}
```

Now, the compiler can tell when we are trying to add an element of the wrong type, and we don't have to worry about getting the wrong type of element out of the collection at run time:
只会在 compile time 的时候看一下，run-time根本不会看，因为run-time那个 T (String, in this case) 已经被抹去了

Generic methods



```
<T> void print(Collection<T> c) {
    for (T x : c) {
        println("value: " + x);
    }
}

Collection<Integer> c;
print(c); // equivalent to this.<Integer>print(c);
```

Subtyping

`LList<String> <: Collection<String>` 合法

`LList<String> <: LList<Object>` 不合法

因为 `HashMap<Object, String> hmap = new HashMap<Integer, String>();` 不合法

Wildcards

`LList<T>` <: `LList<?>`

This means that a method can provide a caller with a list of any type without the client knowing what is really stored in the list; the client can get elements from the list but cannot change the list:

```
LList<?> f() {
  LList<Integer> i = new LList();
  i.add(2);
  i.add(3);
  i.add(5);
  return i;
}

// in caller
LList<?> lst = f();
lst.add(7); // illegal: type ? not known
for (Object o : lst) {
  println(o);
}
```

Note that the type of the elements iterated over is not really known either, but at least we know that the type hidden by ? is a subtype of `Object`. So it is type-safe to declare the variable `o` as an `Object`.

extends

`Collection<? extends Animal>`

we can iterate over the collection and extract `Animal`s rather than just `Object`s.

```
Collection<? extends Animal> c = new LList<Rhino>();
for (Animal a : c) {
  // use a as Animal here
}
```

Limitations

all actual type parameters are erased at run time
we can't instantiate Collection of primitive types, e.g. Collection <int>
It can't do instanceof because T is gone, it doesn’t know what it is

Accessing Type Operations

这是一个所有类 call sort 的时候都应该 implement 的一个类

```
interface Comparator<T> {
        /** Compares x and y. Return 0 if x and y are equal, a negative
number if x < y, and a positive number if x > y. */
        int compareTo(T x, T y);
}
```

比如说我们有一个 generic array class，里面有一个函数叫 sort，这个 sort 的 cmp 函数应该无论 T 是什么，都在当 x>y 时 return 1，就好像上面的 spec 写的一样
为什么要这样写呢？因为我们这个sort函数要能够sort各种各样的数据类型，但是不同的数据类型，比较的原则应该是不同的（比如说String和int肯定就不同），我们不能在这个sort函数里面给所有不同sort应该支持的数据类型，都各自写一个sort函数。所以我们需要找一个这些数据结构通用的“比较器”。

```
/** Sort the array a in ascending order using cmp to define the ordering
on the elements. */
<T> sort(T[] a, Comparator<T> cmp) {
        ...
        if (cmp.compareTo(a[i], a[j]) > 0) {
                ...
        }
        ...
}
```

于是乎，对于不同的 class，这个 cmp 到底怎么写，就成了 sort 是否成功的关键（假定sort函数是正确的情况下）
应用的实例如下：

```
class SCmp implements Comparator<String> {
        @Override
        public int compareTo(String x, String y) {
                return x.compareTo(y);
        }
}

String[] a = {"z", "Y", "x"};
sort(a, new SCmp());
```

To ensure that the type `T` has such an operation, we specify in the class declaration that `T extends Comparable<T>`, where `Comparable<T>` is the generic interface that has the method compareTo