(/)

Optimización del rendimiento de HashMap

Última modificación: 28 de febrero de 2021

por Michał Dąbrowski (https://www.baeldung.com/author/michaldabrowski/) (https://www.baeldung.com/author/michaldabrowski/)

Colecciones de Java (https://www.baeldung.com/category/java/java-collections/)

Mapa de Java (https://www.baeldung.com/tag/java-map/)

1. Introducción

HashMap es una estructura de datos poderosa que tiene una amplia aplicación, especialmente cuando se necesita un tiempo de búsqueda rápido. Sin embargo, si no prestamos atención a los detalles, puede volverse subóptimo.

En este tutorial, veremos cómo hacer *HashMap lo* (/java-hashmap) más rápido posible.

El tiempo constante optimista de recuperación de elementos de *HashMap* (*O (1)*) proviene del poder del hash. Para cada elemento, *HashMap* calcula el código hash y coloca el elemento en el depósito asociado con ese código hash. Debido a que los objetos no iguales pueden tener los mismos códigos hash (un fenómeno llamado colisión de códigos hash), los depósitos pueden aumentar de tamaño.

El cubo es en realidad una simple lista enlazada. Encontrar elementos en la lista vinculada no es muy rápido (*O (n)*) pero eso no es un problema si la lista es muy pequeña. Los problemas comienzan cuando tenemos muchas colisiones de códigos hash, por lo que en lugar de una gran cantidad de cubos pequeños, tenemos una pequeña cantidad de cubos grandes.

En el peor de los casos, en el que ponemos todo dentro de un cubo, nuestro *HashMap* se degrada a una lista vinculada. En consecuencia, en lugar del tiempo de búsqueda de *O (1)* , obtenemos un *O (n)* muy insatisfactorio .

3. Árbol en lugar de *LinkedList*

A partir de Java 8, una optimización (https://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/a006fa0a9e8f/src/share/classes/java/util/HashMap.java#l143) está incorporada en *HashMap* : **cuando los depósitos se vuelven demasiado grandes, se transforman en árboles, en lugar de listas vinculadas.** Eso trae el tiempo pesimista de *O (n)* a *O (log (n))* , que es mucho mejor. **Para que eso funcione, las claves de *HashMap* deben implementar la interfaz *Comparable* (/java-comparator-comparable) .**

Esa es una solución agradable y automática, pero no es perfecta. *O (log (n))* sigue siendo peor que el tiempo constante deseado, y la transformación y el almacenamiento de árboles requiere más energía y memoria.

4. Mejor implementación de *hashCode*

Hay dos factores que debemos tener en cuenta al elegir una función hash: la calidad de los códigos hash producidos y la velocidad.

4.1. Medición de la calidad de *hashCode*

Los códigos hash se almacenan dentro de las variables *int* , por lo que el número de hash posibles se limita a la capacidad del tipo *int* . Debe ser así porque los hashes se utilizan para calcular índices de una matriz con cubos. Eso significa que también hay una cantidad limitada de claves que podemos almacenar en un *HashMap* sin colisión de hash.

Para evitar colisiones el mayor tiempo posible, queremos distribuir los hash de la forma más uniforme posible. En otras palabras, queremos lograr una distribución uniforme. Eso significa que cada valor de código hash tiene las mismas posibilidades de ocurrir que cualquier otro.

Del mismo modo, un método *hashCode* incorrecto tendría una distribución muy desequilibrada. En el peor de los casos, siempre devolvería el mismo número.

4.2. Por defecto *de objeto* 's *hashCode*

En general, no deberíamos usar el método *hashCode del objeto* predeterminado porque no queremos usar la identidad del objeto (/java-map-key-byte-array#4-meaningful-equality) en el método *equals* . Sin embargo, en ese escenario muy poco probable en el que realmente queremos usar la identidad del objeto para las claves en un *HashMap* , la función *hashCode* predeterminada funcionará bien. De lo contrario, querremos una implementación personalizada. (/java-map-key-byte-array#4-meaningful-equality)

4.3. *HashCode* personalizado

Por lo general, queremos anular el método *equals* , y luego también necesitamos anular *hashCode* (/java-equals-hashcode-contracts) . A veces, podemos aprovechar la identidad específica de la clase y hacer fácilmente un método (/java-hashcode)*hashCode* (/java-hashcode) muy rápido .

Digamos que la identidad de nuestro objeto se basa puramente en su *ID de* número entero . Entonces, podemos usar esta *identificación* como una función hash:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    MemberWithId that = (MemberWithId) o;

    return id.equals(that.id);
}

@Override
public int hashCode() {
    return id;
}
```

Será extremadamente rápido y no producirá colisiones. **Nuestro *HashMap* se comportará como si tuviera una clave entera en lugar de un objeto complejo.**

La situación se complicará más si tenemos más campos que debemos tener en cuenta. Digamos que queremos basar la igualdad tanto en *id* como en *name* :

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    MemberWithIdAndName that = (MemberWithIdAndName) o;

    if (!id.equals(that.id)) return false;
    return name != null ? name.equals(that.name) : that.name == null;
}
```

Ahora, necesitamos combinar de alguna manera los valores hash de *id* y *name* .

Primero, obtendremos el hash de *id* igual que antes. Luego, lo multiplicaremos por un número cuidadosamente elegido y agregaremos el hash del *nombre* :

```
@Override
public int hashCode() {
    int result = id.hashCode();
    result = PRIME * result + (name != null ? name.hashCode() : 0);
    return result;
}
```

Cómo elegir ese número no es una pregunta fácil de responder suficientemente. Históricamente, el número más popular era *31.* Es primo, da como resultado una buena distribución, es pequeño y se puede optimizar al multiplicarlo mediante una operación de desplazamiento de bits:

```
31 * i == (i << 5) - i
```

Sin embargo, ahora que no necesitamos luchar por cada ciclo de la CPU, se pueden usar algunos números primos más grandes. Por ejemplo, *524287* también se puede optimizar:

```
524287 * i == i << 19 - i
```

Y puede proporcionar un hash de mejor calidad, lo que reduce las posibilidades de colisión. Tenga en cuenta que estas **optimizaciones de desplazamiento de bits las realiza automáticamente la JVM** , por lo que no necesitamos ofuscar nuestro código con ellas.

4.4. Clase de utilidad de *objetos*

El algoritmo que acabamos de implementar está bien establecido y, por lo general, no es necesario volver a crearlo a mano cada vez. En su lugar, podemos usar el método auxiliar proporcionado por la clase *Objects* :

```
@Override
public int hashCode() {
    return Objects.hash(id, name);
}
```

Bajo el capó, utiliza exactamente el algoritmo descrito anteriormente con el número *31* como multiplicador.

4.5. Otras funciones hash

Hay muchas funciones hash que brindan una probabilidad de colisión menor que la descrita anteriormente. El problema es que computacionalmente son más pesados y, por lo tanto, no brindan la ganancia de velocidad que buscamos.

Si por alguna razón realmente necesitamos calidad y no nos importa mucho la velocidad, podemos echar un vistazo a la clase *Hashing* de la biblioteca de Guava (/guava-guide) :

```
@Override
public int hashCode() {
    HashFunction hashFunction = Hashing.murmur3_32();
    return hashFunction.newHasher()
      .putInt(id)
      .putString(name, Charsets.UTF_8)
      .hash().hashCode();
}
```

Es importante elegir una función de 32 bits porque de todos modos no podemos almacenar hash más largos.

5. Conclusión

El *HashMap* de Java moderno es una estructura de datos potente y bien optimizada. Sin embargo, su rendimiento puede empeorar con un método *hashCode* mal diseñado . En este tutorial, analizamos posibles formas de hacer hash rápido y efectivo.

Como siempre, los ejemplos de código de este artículo están disponibles en GitHub (https://github.com/eugenp/tutorials/tree/master/java-collections-maps-3) .

¿Aprendiendo a construir su API **con Spring** ?

Ingrese su dirección de correo electrónico

>> Obtenga el eBook

Acceso (https://www.baeldung.com/wp-login.php?redirect_to=https%3A%2F%2Fwww.baeldung.com%2Fjava-hashmap-optimize-performance)

Be the First to Comment!

{} [+]

0 COMENTARIOS

'freestar.com/?ce=branding&utm_name=baeldung_adhesion)

CATEGORIAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
LADO DEL CLIENTE HTTP (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (/JAVA-TUTORIAL)
TUTORIAL DE JACKSON JSON (/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (/HTTPCLIENT-GUIDE)
DESCANSO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL DE PERSISTENCIA DE PRIMAVERA (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON SPRING (/SECURITY-SPRING)

SOBRE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJOS (/TAG/ACTIVE-JOB/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBE PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANÚNCIESE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)