

Effective Java Study

Woowacourse_study 4th



Item 63 by 알파

결론

그냥 StringBuilder 쓰세요

왜..?

“+=” 연산이 매우매우 느리거든요

디버깅의 세계로..

```
public class StringAdd {  
    public static void main(String[] args) {  
        String now = "Alpha";  
        for (int i = 0; i < 10; i++) {  
            now += "WoowaCourse";  
        }  
        System.out.print(now);  
    }  
}
```

디버깅의 세계로..

▼ now	"Alpha" (id=20)
└─ coder	0
└─ hash	0
└─ > value	(id=26)

디버깅의 세계로..

```
2950 public static String valueOf(Object obj) {  
2951     return (obj == null) ? "null" : obj.toString();  
2952 }
```







이 때 obj는 “Alpha”

디버깅의 세계로..

```
2806 public String toString() {  
2807     return this;  
2808 }
```

이 때 this는 “Alpha”

디버깅의 세계로..

>  valueOf() returned	"Alpha" (id=21)
 args	String[0] (id=19)
✓  now	"Alpha" (id=21)
 coder	0
 hash	0
>  value	(id=26)

Now의 id가 변했다!

디버깅의 세계로..

▼ ⓘ now	"Alpha" (id=20)
🔍 coder	0
🔍 hash	0
> 🔍 value	(id=26)

> 📄 valueOf() returned	"Alpha" (id=21)
🔍 args	String[0] (id=19)
▼ ⓘ now	"Alpha" (id=21)
🔍 coder	0
🔍 hash	0
> 🔍 value	(id=26)

Now의 id가 변했다!

잠시 정리

단순히 로컬 변수인 `now`를 내용은 `Alpha`로서 같지만 다른
`String` 객체로 바꿈

튼금없이 StringBuilder?

```
127• @HotSpotIntrinsicCandidate
128 public StringBuilder(String str) {
129     super(str.length() + 16);
130     append(str);
131 }
```

Capacity가 21인 새로운 StringBuilder의 인스턴스를 생성,
이 때 str은 “Alpha”

튼금없이 StringBuilder?

```
176  @Override
177  @HotSpotIntrinsicCandidate
178  public StringBuilder append(String str) {
179      super.append(str);
180      return this;
181  }
```

튼금없이 StringBuilder?

```
533 public AbstractStringBuilder append(String str) {  
534     if (str == null) {  
535         return appendNull();  
536     }  
537     int len = str.length();  
538     ensureCapacityInternal(count + len);  
539     putStringAt(count, str);  
540     count += len;  
541     return this;  
542 }
```

이 때 str은 역시 “Alpha”

튼금없이 StringBuilder?

```
public AbstractStringBuilder append(String str) {  
    if (str == null) {  
        return appendNull();  
    }  
    int len = str.length();  
    ensureCapacityInternal(count + len);  
    putStringAt(count, str);  
    count += len;  
    return this;  
}
```

내부적으로 capacity를 늘려줌

튼금없이 StringBuilder?

▼ • this	StringBuilder (id=41)
▲ coder	0
▲ count	0
> ▲ value	(id=44)

Count는 새로만든 sb의 필드 정도로 이해하자

튼금없이 StringBuilder?

```
168     private void ensureCapacityInternal(int minimumCapacity) {  
169         // overflow-conscious code  
170         int oldCapacity = value.length >> coder;  
171         if (minimumCapacity - oldCapacity > 0) {  
172             value = Arrays.copyOf(value,  
173                 newCapacity(minimumCapacity) << coder);  
174         }  
175     }
```

oldCapacity = 21, minimumCapacity = 5

핵심 로직

```
533 public AbstractStringBuilder append(String str) {  
534     if (str == null) {  
535         return appendNull();  
536     }  
537     int len = str.length();  
538     ensureCapacityInternal(count + len);  
539     putStringAt(count, str);  
540     count += len;  
541     return this;  
542 }
```

oldCapacity = 21, minimumCapacity = 5

핵심 로직

```
1663 private final void putStringAt(int index, String str) {  
1664     if (getCoder() != str.coder()) {  
1665         inflate();  
1666     }  
1667     str.getBytes(value, index, coder);  
1668 }
```

Index = 0, str = "Alpha"
조건문의 getCoder()의 리턴값은 0

핵심 로직

```
1663     private final void putStringAt(int index, String str) {  
1664         if (getCoder() != str.coder()) {  
1665             inflate();  
1666         }  
1667         str.getBytes(value, index, coder);  
1668     }
```

Index = 0, coder = 0

핵심 로직

▼ • this	StringBuilder (id=41)
▲ coder	0
▲ count	0
> ▲ value	(id=44)
⊙ index	0
▼ ⊙ str	"Alpha" (id=22)
▣ coder	0
▣ hash	0
> ▣ value	(id=26)

파라미터인 value는 sb인 this가 가지는 byte 배열

핵심 로직

```
3190 void getBytes(byte dst[], int dstBegin, byte coder) {  
3191     if (coder() == coder) {  
3192         System.arraycopy(value, 0, dst, dstBegin << coder, value.length);  
3193     } else { // this.coder == LATIN && coder == UTF16  
3194         StringLatin1.inflate(value, 0, dst, dstBegin, value.length);  
3195     }  
3196 }
```

조건문을 통과하기 때문에
어떤 copy를 할 것만 같은 메소드로 진입

잠시 정리

**Capacity가 21짜리인 어떤 StringBuilder를 만든 이후에
원래 문자열을 복사하기 직전**

핵심 로직

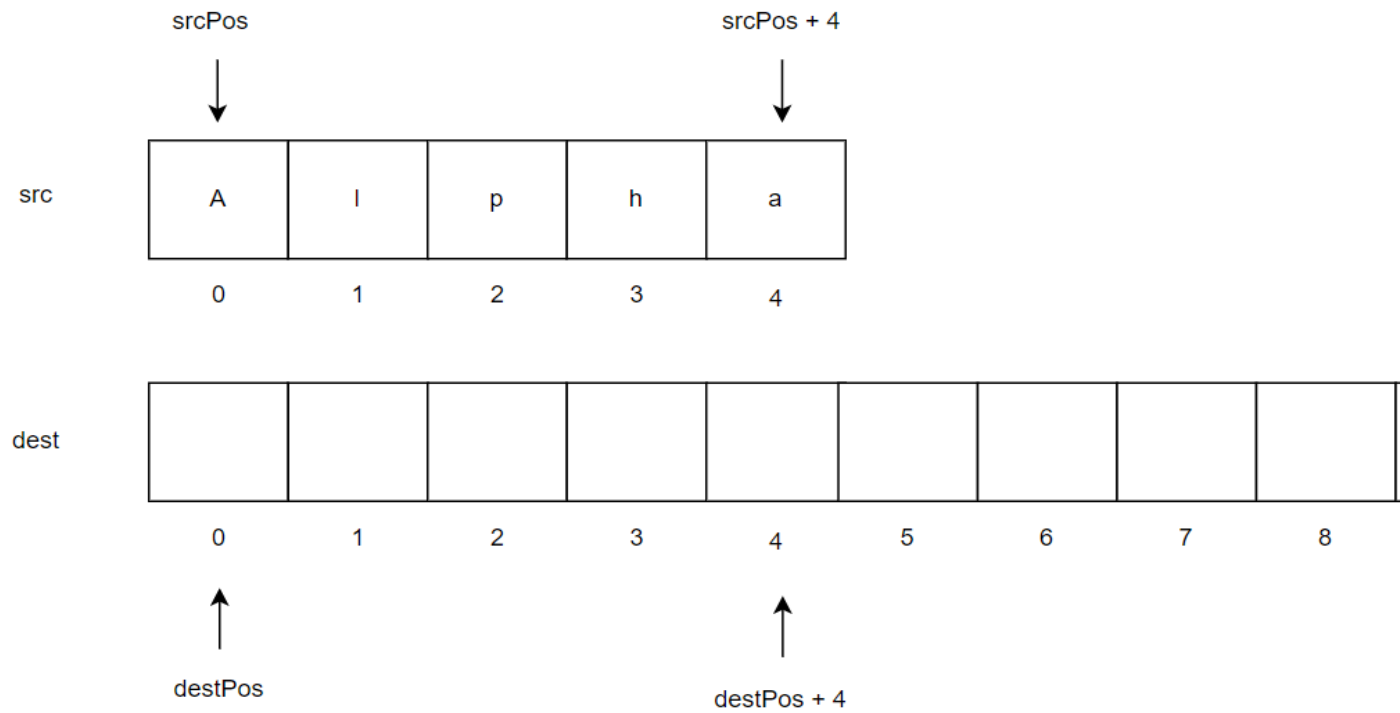
`arraycopy`

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

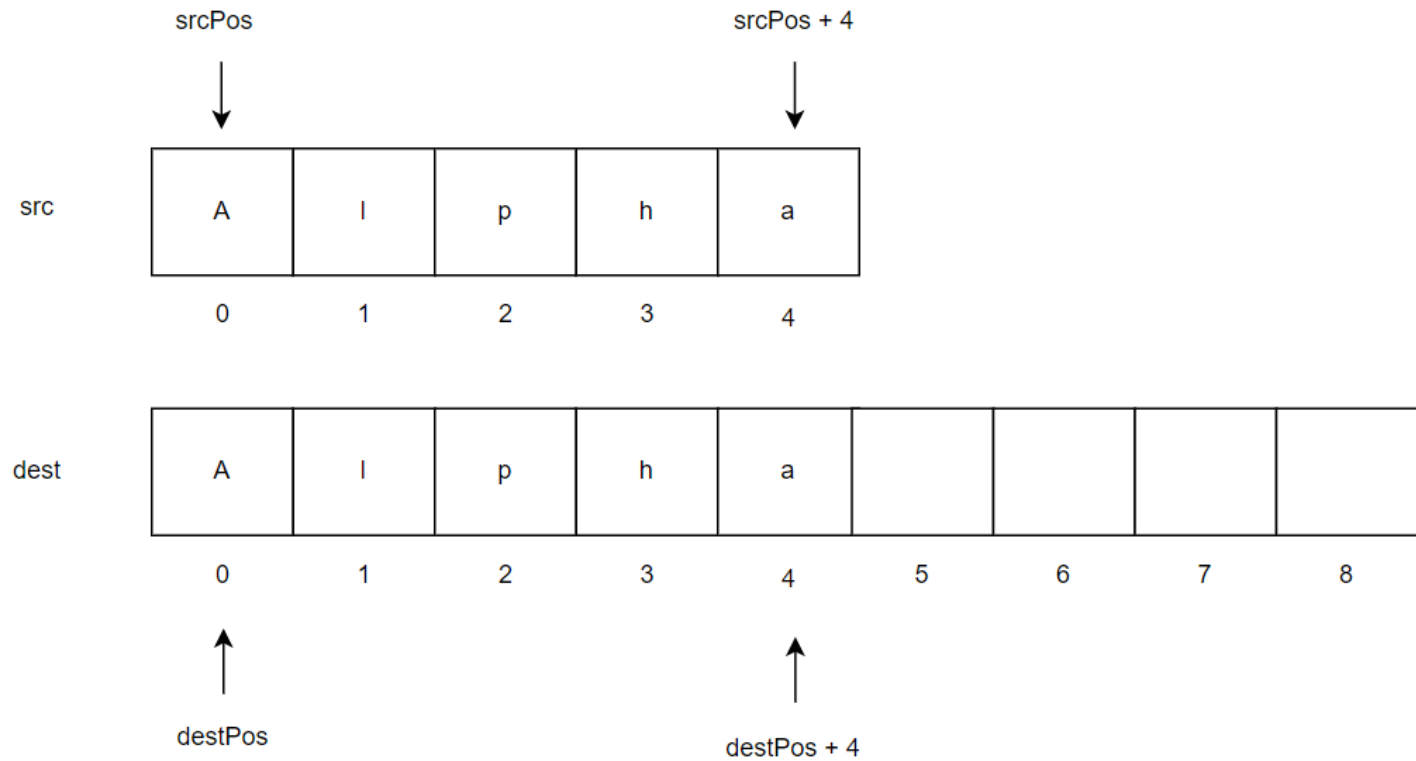
Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

**원본 배열인 src의 srcPos에서 length개의 원소를 복사본 배열인 dest의 destPos부터 추가,
이 때 원소 구간은 [srcPos, srcPos + length)**

핵심 로직



핵심 로직



핵심 로직

Copy에 드는 시간복잡도 $\rightarrow O(N)$

Name	Value
▼ value	(id=26)
▶ [0]	65
▶ [1]	108
▶ [2]	112
▶ [3]	104
▶ [4]	97
▼ ● dst	(id=44)
▶ [0]	65
▶ [1]	108
▶ [2]	112
▶ [3]	104
▶ [4]	97
▶ [5]	0

핵심 로직

```
533 public AbstractStringBuilder append(String str) {  
534     if (str == null) {  
535         return appendNull();  
536     }  
537     int len = str.length();  
538     ensureCapacityInternal(count + len);  
539     putStringAt(count, str);  
540     count += len;  
541     return this;  
542 }
```

아까 count는 sb의 필드라고 생각했는데,
복사 이후 복사 길이만큼 더해준다
따라서, count는 이때까지 복사한 문자열의 길이를
count한 의미라고 추론이 가능하다!

핵심 로직

```
176  @Override
177  @HotSpotIntrinsicCandidate
178  public StringBuilder append(String str) {
179      super.append(str);
180      return this;
181  }
```

최종적으로 원래 문자열의 내용을 복사한 sb가 리턴

잠시 정리

여기까지 원래 문자열 `now`를 `StringBuilder`에 복사하고 `sb`를 리턴했다

잠시 정리

리턴된 `StringBuilder`에 더하고자 하는 문자열을 또 복사한다

시간복잡도 $\rightarrow O(N)$ (~~상수고려한다면 $2N$~~)

잠시 정리

Name	Value
no method return value	
• this	StringBuilder (id=41)
• coder	0
• count	5
> • value	(id=44)
• str	"WoowaCourse" (id=49)
• coder	0
• hash	0
> • value	(id=50)

“Alpha”를 복사한 이후 조사식

최종적으로..

**두 문자열을 concat한 StringBuilder의 toString()을 통해
원래 문자열이었던 now가 “AlphaWoowaCourse”를 가리킴**

최종적으로..

args	String[0] (id=19)
now	"AlphaWoowaCourse" (id=52)
coder	0
hash	0
value	(id=53)

최종적으로..

**원래 문자열의 길이를 s1, 더하고자 하는 문자열의 길이가 s2일
때, $O(\max(s1, s2))$**

N개의 문자열을 더한다면 $\rightarrow O(N * \text{Longest Size of String})$

그래서 결론은..?

StringBuilder써서 그냥 append쓰자..?

StringBuilder의 append도 $O(N)$ 아닌가?

Amortized Time Complexity

어떤 연산을 했을 때, 일정한 시간 복잡도가 나오는 것이 아니라,
여러 케이스의 시간 복잡도가 나올 때 현재까지 나온 시간 복잡
도의 합을 현재 시점을 기준으로 나눠서 계산하는 것

Amortized TC **VS** Average TC

Amortized : 입력의 크기에 상관없이 최선인 경우, 최악인 경우가 같다

Average : 입력의 크기에 따라 최선의 경우, 최악의 경우가 다르니까 확률적으로 평균 입력 크기일 때를 구한다

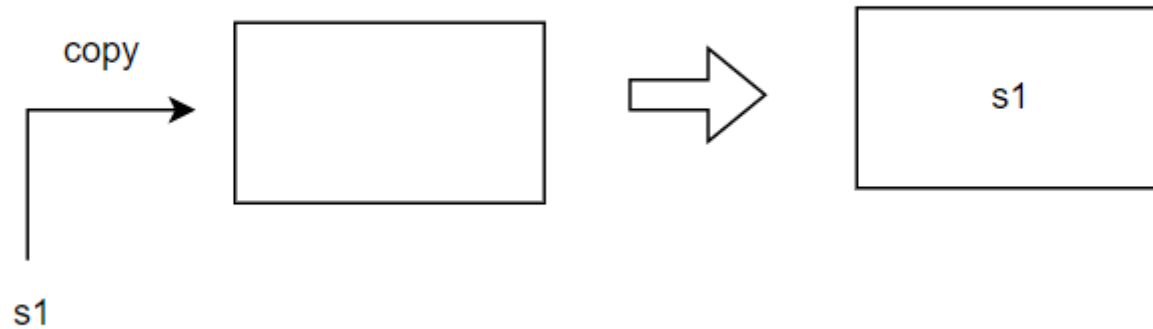
Amortized Time Complexity

총 N개의 문자열을 전부 append하는 상황을 생각해보자
이 때, 각각의 문자열 길이는 s_1, s_2, \dots 라고 하자

Amortized Time Complexity

첫 append에서는 capacity가 넉넉하기 때문에 s1만큼의 복사가 일어난다 -> $O(s1)$

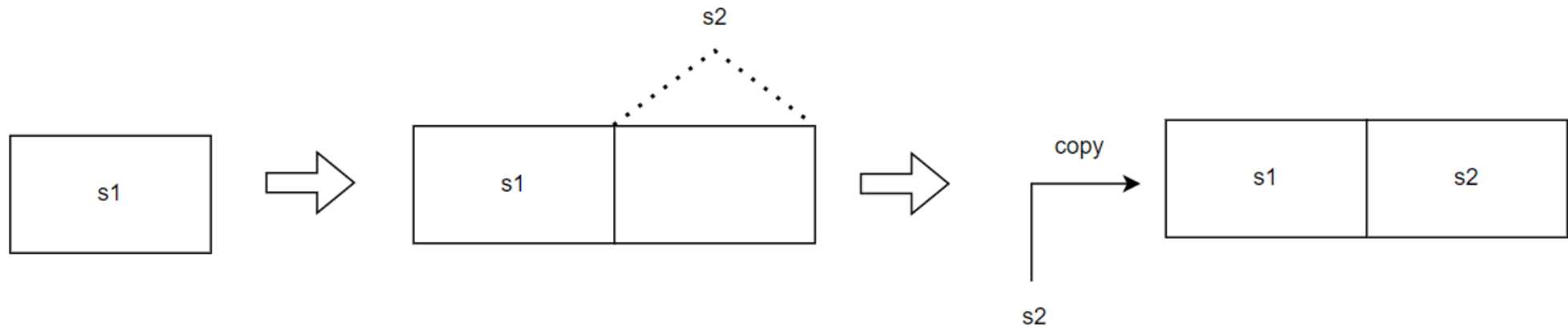
Amortized Time Complexity



Amortized Time Complexity

두번째 append에서는 capacity가 부족하기 때문에 $s1 + s2$ 만큼 capacity를 늘린 후, s2를 복사한다 $\rightarrow O(s1 + s2)$

Amortized Time Complexity



Amortized Time Complexity

세번째 append에서는 capacity가 부족하기 때문에 $s1 + s2 + s3$ 만큼 capacity를 늘린 후, $s3$ 를 복사한다 $\rightarrow O(s1 + s2 + s3)$

Amortized Time Complexity

N번째 append가 다 끝나고 나면, $O(\sum_{i=1}^n s_i)$

전체 복잡도의 합 = $O(s_1) + O(s_1 + s_2) + O(s_1 + s_2 + s_3) + \dots$

Amortized Time Complexity

$$\text{Amortized 복잡도} = \frac{nS_1 + (n-1)S_2 + (n-2)S_3 + \dots + S_n}{S_1 + S_2 + \dots + S_n} = O(n)$$

실험 결과

```
5 public static void main(String[] args) {
6     long plusStart = System.currentTimeMillis();
7     String temp1 = "";
8     for (int i = 0; i < 50000; i++) {
9         temp1 += "a";
10    }
11    long plusEnd = System.currentTimeMillis();
12    System.out.println("+= 실행 시간 : " + (plusEnd - plusStart) + "ms");
13    long appendStart = System.currentTimeMillis();
14    StringBuilder temp2 = new StringBuilder();
15    for (int i = 0; i < 50000; i++) {
16        temp2.append("a");
17    }
18    long appendEnd = System.currentTimeMillis();
19    System.out.println("append 실행 시간 : " + (appendEnd - appendStart) + "ms");
20 }
```

Run: Main ×

```
"C:\Program Files (x86)\Java\jdk1.8.0_311\bin\java.exe" ...
+= 실행 시간 : 3279ms
append 실행 시간 : 1ms

Process finished with exit code 0
```

추론

분명히 `StringBuilder`의 `append`의 시간복잡도는 계산 시 $O(n)$ 이었는데, 어떻게 저렇게 빠른걸까?

가설 : `System.arraycopy`가 사실 $O(n)$ 이 아니라, 더 빠르게 동작한다

추론

실제로 `System.arraycopy()`는 native코드를 통해
동작하기 때문에 가능성이 있다
그러나, 정확한 시간복잡도를 알지는 못했다

차이는 어디에

**String의 += 연산에서는 연산이 끝난 이후 StringBuilder가
toString으로 원래 문자열에 할당이 되는데, 이 부분이 유일한
차이**

차이는 어디에

**이 때 char 배열을 선언하거나 경우에 따라서는 내부적으로
system.arraycopy를 쓴다.**

**사실상 char 배열을 만드는데 필요한 시간이 핵심적인 차이가
된다고 의심해볼 수 있다**

결론

그냥 StringBuilder 쓰세요

+ append는 $O(n)$ 같아보이지만, 실제로는 더 빠르다!

References

Joshua Bloch, 『Effective Java 3/E』, 이복연 역 (서울 : 인사이트, 2018), pp. 366 – 367

References

<https://medium.com/@satorusozaki/amortized-time-in-the-time-complexity-of-an-algorithm-6dd9a5d38045>

<https://stackoverflow.com/questions/18638743/is-it-better-to-use-system-arraycopy-than-a-for-loop-for-copying-arrays>

<https://stackoverflow.com/questions/2772152/why-is-system-arraycopy-native-in-java>

<https://codingdog.tistory.com/entry/java-string-%EC%97%B0%EC%82%B0-%EC%96%B4%EB%96%BB%EA%B2%8C->

<https://codingdog.tistory.com/entry/java-string-%EC%97%B0%EC%82%B0-%EC%96%B4%EB%96%BB%EA%B2%8C-%EB%8F%99%EC%9E%91%ED%95%98%EB%8A%94%EC%A7%80->

E.O.D



Item 63 by 알파