技术分享会

JVM - 引用类型

黃耿霖 20190404

为什么要讲JVM?

什么是内存泄露和内存溢出？

内存溢出是内存爆了，内存空间不足；内存泄漏会导致内存溢出。

内存溢出

程序的内存堆

程序需要开辟
的新内存空间

程序的内存堆

内存不足以开辟新的内存空间，就会出现**内存溢出**的异常

什么是内存泄露？

起初程序的运行内存情况

“为什么我的程序起初跑得好好的，上线运行了一段时间就爆内存了？”

有些内存本来就应该回收但没有回收，还留着内存里占空间

对象引用的内存在没有用之后没有被释放，称为“内存泄漏”

为什么内存不会被释放?

引用

- 强引用
- 软引用
- 弱引用
- 幽灵引用

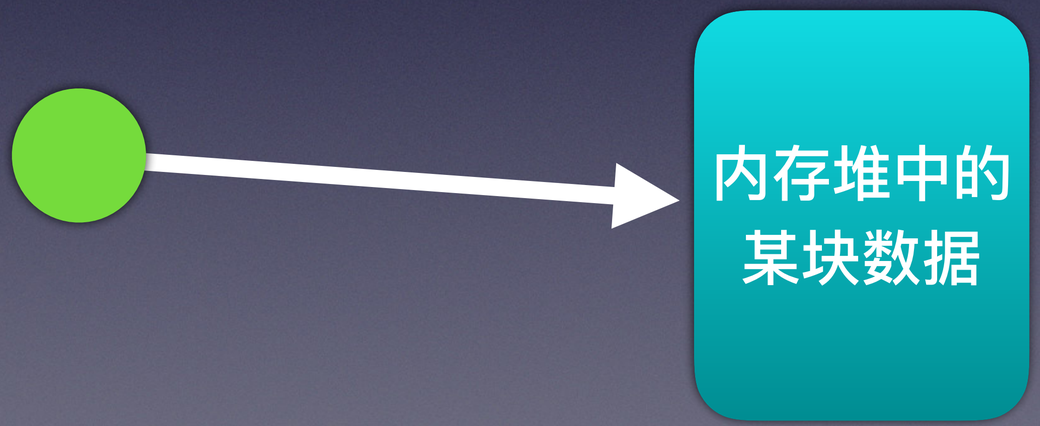
- 强引用

被强引用引用的内存在垃圾回收时不会被回收。

什么是垃圾回收（GarbageCollection）？

垃圾回收的算法

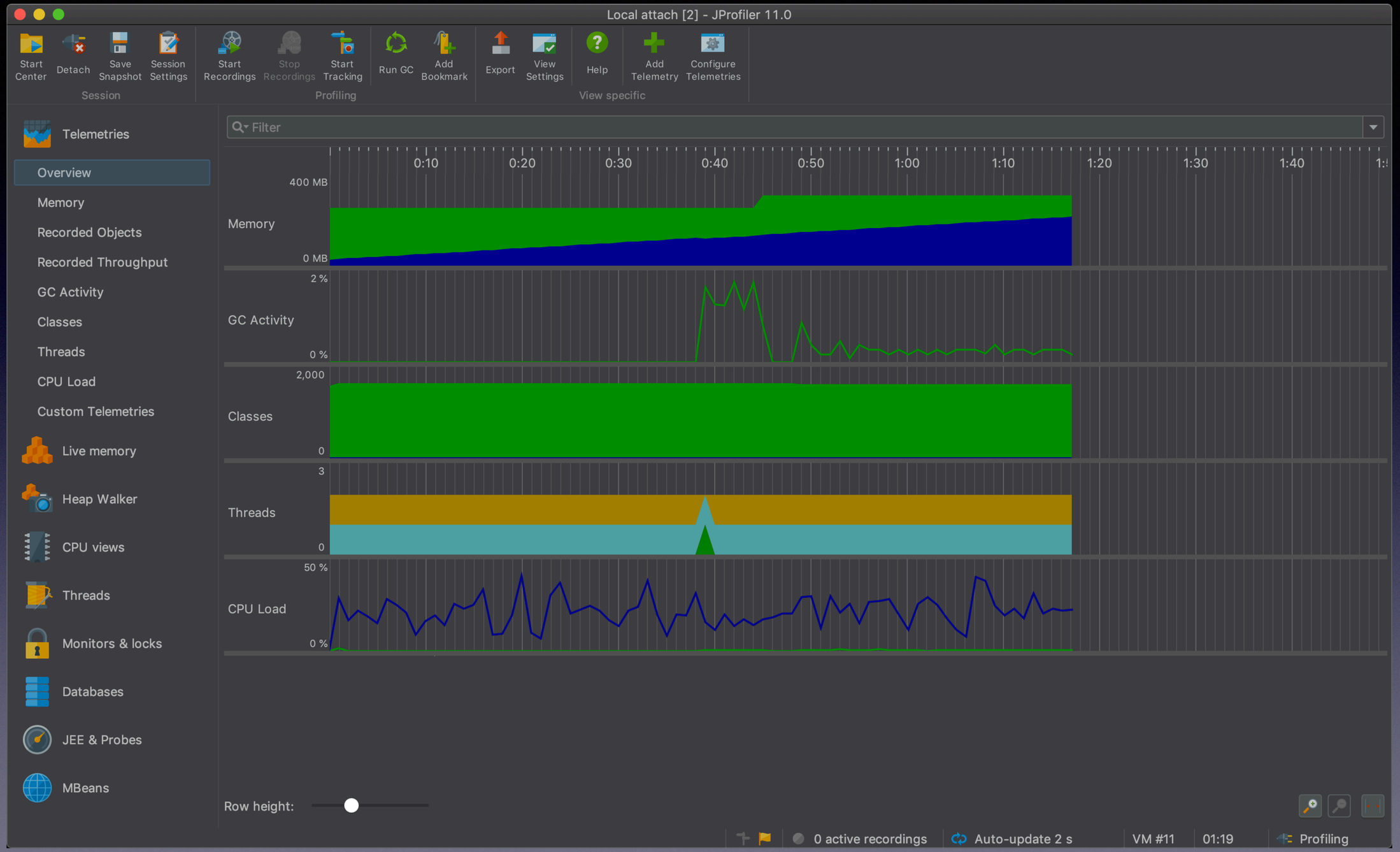
被强引用引用的内存**在垃圾回收时不会被回收**。

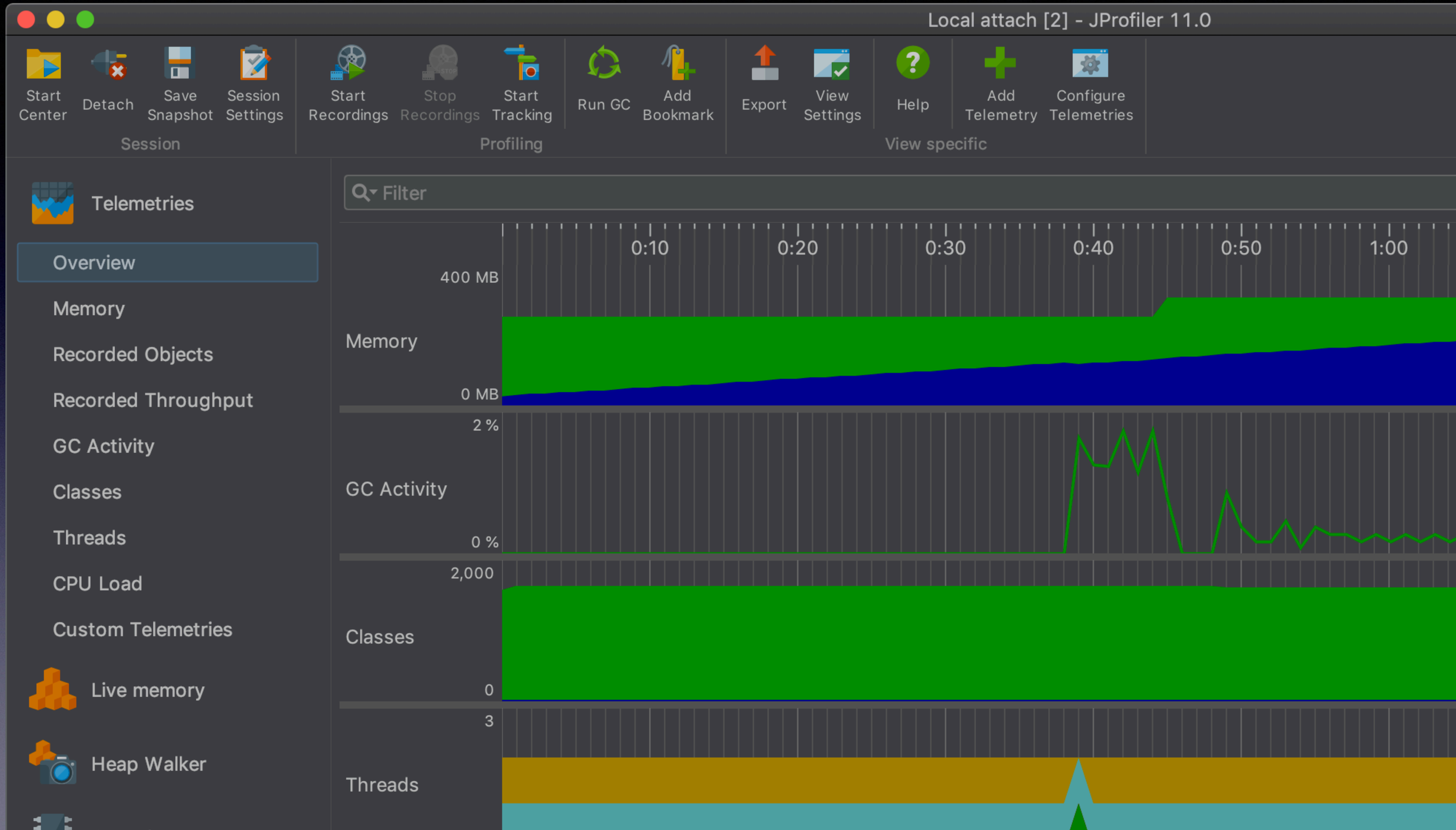


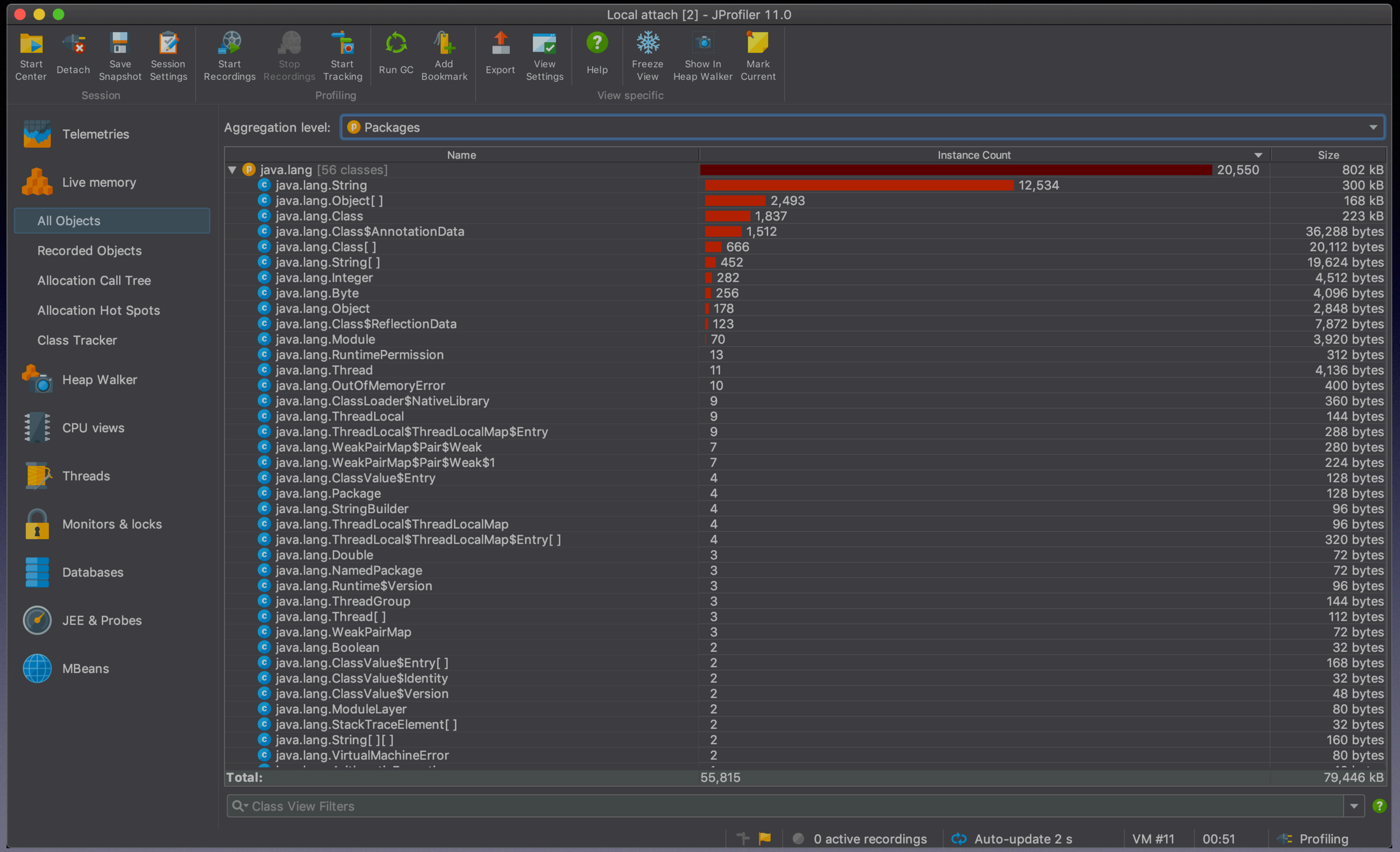
Demo之前

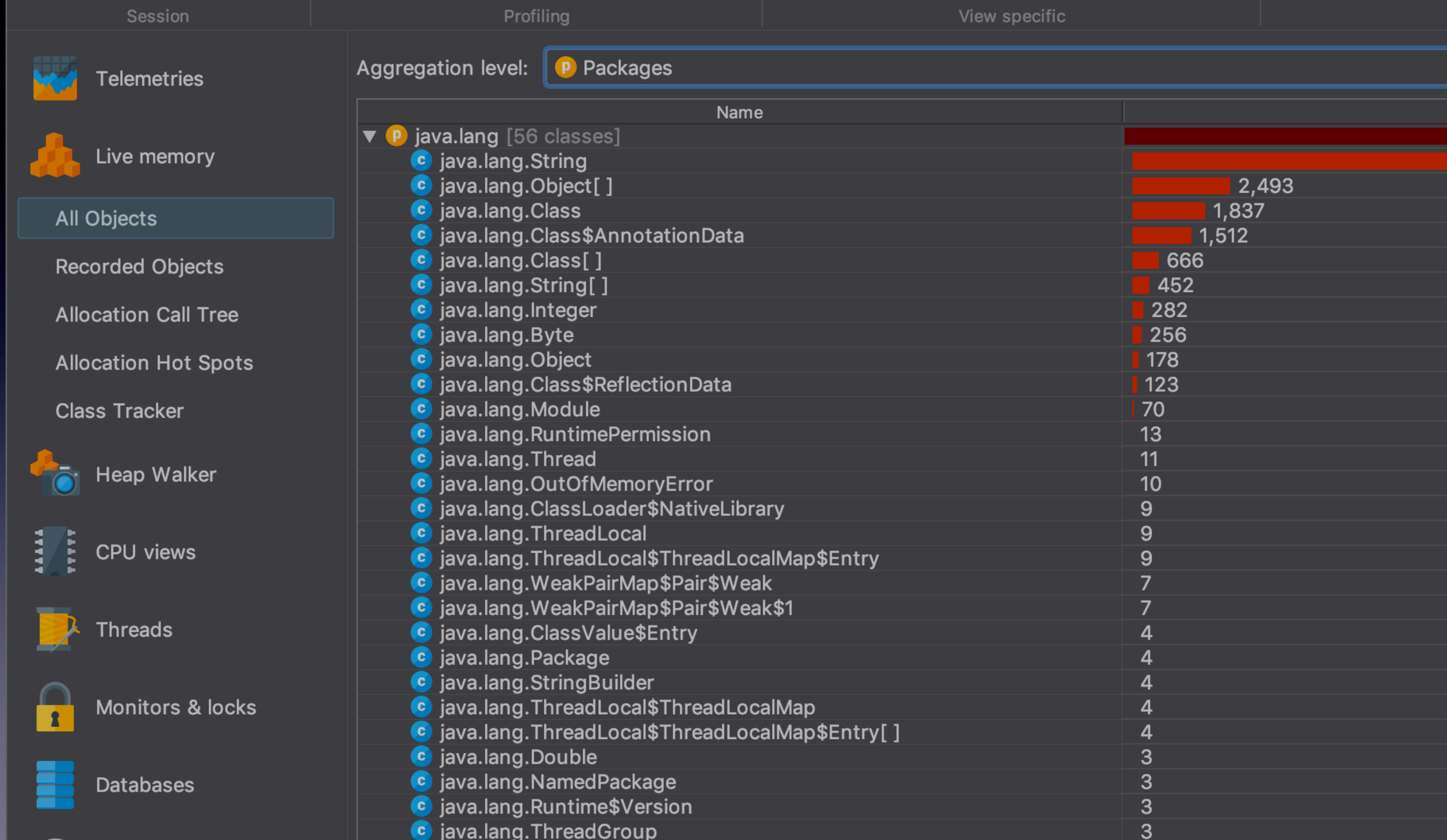
JProfiler

https://www.ej-technologies.com/products/jprofiler/overview.html


Local attach [2] - JProfiler 11.0
Start Center
Detach
Save Snapshot
Session Settings
Session
Start Recordings
Stop Recordings
Start Tracking
Profiling
Run GC
Add Bookmark
Export
View Settings
Help
Add Telemetry
Configure Telemetries
View specific
Telemetries
Overview
Memory
Recorded Objects
Recorded Throughput
GC Activity
Classes
Threads
CPU Load
Custom Telemetries
Live memory
Heap Walker
CPU views
Threads
Monitors & locks
Databases
JEE & Probes
MBeans
Filter
0:10
0:20
0:30
0:40
0:50
1:00
1:10
1:20
1:30
1:40
400 MB
Memory
0 MB
2 %
GC Activity
0 %
2,000
Classes
0
3
Threads
0
50 %
CPU Load
0 %
Row height:
0 active recordings
Auto-update 2 s
VM #11
01:19
Profiling


Local attach [2] - JProfiler 11.0
Start Center
Detach
Save Snapshot
Session Settings
Session
Start Recordings
Stop Recordings
Start Tracking
Profiling
Run GC
Add Bookmark
Export
View Settings
Help
Add Telemetry
Configure Telemetries
View specific
Telemetries
Overview
Memory
Recorded Objects
Recorded Throughput
GC Activity
Classes
Threads
CPU Load
Custom Telemetries
Live memory
Heap Walker
Filter
0:10
0:20
0:30
0:40
0:50
1:00
400 MB
Memory
0 MB
2 %
GC Activity
0 %
2,000
Classes
0
3
Threads


Local attach [2] - JProfiler 11.0
Start Center
Detach
Save Snapshot
Session Settings
Session
Start Recordings
Stop Recordings
Start Tracking
Run GC
Add Bookmark
Profiling
Export
View Settings
Help
Freeze View
Show In Heap Walker
Mark Current
View specific
Telemetries
Live memory
All Objects
Recorded Objects
Allocation Call Tree
Allocation Hot Spots
Class Tracker
Heap Walker
CPU views
Threads
Monitors & locks
Databases
JEE & Probes
MBeans
Aggregation level: Packages
Name
Instance Count
Size
java.lang [56 classes] 20,550 802 kB
java.lang.String 12,534 300 kB
java.lang.Object[ ] 2,493 168 kB
java.lang.Class 1,837 223 kB
java.lang.Class$AnnotationData 1,512 36,288 bytes
java.lang.Class[ ] 666 20,112 bytes
java.lang.String[ ] 452 19,624 bytes
java.lang.Integer 282 4,512 bytes
java.lang.Byte 256 4,096 bytes
java.lang.Object 178 2,848 bytes
java.lang.Class$ReflectionData 123 7,872 bytes
java.lang.Module 70 3,920 bytes
java.lang.RuntimePermission 13 312 bytes
java.lang.Thread 11 4,136 bytes
java.lang.OutOfMemoryError 10 400 bytes
java.lang.ClassLoader$NativeLibrary 9 360 bytes
java.lang.ThreadLocal 9 144 bytes
java.lang.ThreadLocal$ThreadLocalMap$Entry 9 288 bytes
java.lang.WeakPairMap$Pair$Weak 7 280 bytes
java.lang.WeakPairMap$Pair$Weak$1 7 224 bytes
java.lang.ClassValue$Entry 4 128 bytes
java.lang.Package 4 128 bytes
java.lang.StringBuilder 4 96 bytes
java.lang.ThreadLocal$ThreadLocalMap 4 96 bytes
java.lang.ThreadLocal$ThreadLocalMap$Entry[ ] 4 320 bytes
java.lang.Double 3 72 bytes
java.lang.NamedPackage 3 72 bytes
java.lang.Runtime$Version 3 96 bytes
java.lang.ThreadGroup 3 144 bytes
java.lang.Thread[ ] 3 112 bytes
java.lang.WeakPairMap 3 72 bytes
java.lang.Boolean 2 32 bytes
java.lang.ClassValue$Entry[ ] 2 168 bytes
java.lang.ClassValue$Identity 2 32 bytes
java.lang.ClassValue$Version 2 48 bytes
java.lang.ModuleLayer 2 80 bytes
java.lang.StackTraceElement[ ] 2 32 bytes
java.lang.String[ ][ ] 2 160 bytes
java.lang.VirtualMachineError 2 80 bytes
Total: 55,815 79,446 kB
Class View Filters
0 active recordings
Auto-update 2 s
VM #11
00:51
Profiling


Session
Profiling
View specific
Telemetries
Live memory
All Objects
Recorded Objects
Allocation Call Tree
Allocation Hot Spots
Class Tracker
Heap Walker
CPU views
Threads
Monitors & locks
Databases
Aggregation level:
Packages
Name
java.lang [56 classes]
java.lang.String
java.lang.Object[ ] 2,493
java.lang.Class 1,837
java.lang.Class$AnnotationData 1,512
java.lang.Class[ ] 666
java.lang.String[ ] 452
java.lang.Integer 282
java.lang.Byte 256
java.lang.Object 178
java.lang.Class$ReflectionData 123
java.lang.Module 70
java.lang.RuntimePermission 13
java.lang.Thread 11
java.lang.OutOfMemoryError 10
java.lang.ClassLoader$NativeLibrary 9
java.lang.ThreadLocal 9
java.lang.ThreadLocal$ThreadLocalMap$Entry 9
java.lang.WeakPairMap$Pair$Weak 7
java.lang.WeakPairMap$Pair$Weak$1 7
java.lang.ClassValue$Entry 4
java.lang.Package 4
java.lang.StringBuilder 4
java.lang.ThreadLocal$ThreadLocalMap 4
java.lang.ThreadLocal$ThreadLocalMap$Entry[ ] 4
java.lang.Double 3
java.lang.NamedPackage 3
java.lang.Runtime$Version 3
java.lang.ThreadGroup 3

Demo

强引用的意义?

存放需要长驻内存、稳定、精确、不丢失的数据

- 强引用我们最常用的引用
- 也是造成内存泄露的主要排查原因

- 软引用

被软引用引用的内存**在内存不足时**才会被回收。

Demo

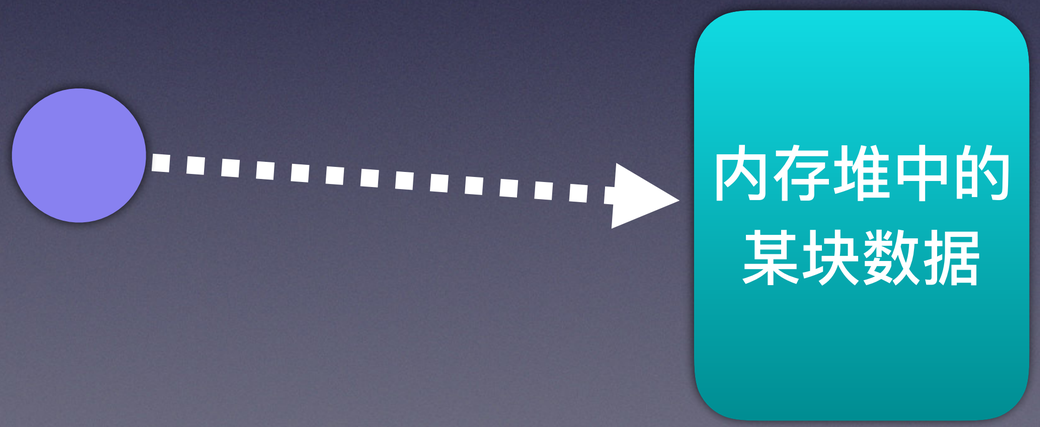
软引用的意义?

存放那些可以优先放在内存的数据，比如图片、缓存数据。

根据读写速度【寄存器】>【内存】>【磁盘】，在内存足够的前提下，
把数据放在内存能提高程序的性能，也可以充分利用内存优势。

- 虚引用

被虚引用引用的内存**在内存回收时**会立马被回收。



Demo

虚引用的意义?

先谈谈finalize()方法

垃圾回收器会特别对待覆盖了finalize()方法的对象。

一般情况下，在垃圾回收期间，一个无法触及的对象会立即被销毁。

不过，覆盖了finalize()方法的对象会被移动到一个队列里，一个独立的线程遍历这个队列，**调用每一个对象的finalize()方法**。在finalize()方法调用结束之后，这些对象才成为真正的垃圾，等待下一轮垃圾回收。

Demo

虚引用的意义?

- 被弱引用引用的对象是可有可无的状态
- 可以进行日志监控、监听重要对象被回收
- 监听系统GC事件。因为虚引用每次GC都会被回收，那么我们就可以通过虚引用来判断GC的频率，如果频率过大，内存使用可能存在问题，才导致了系统GC频繁调用

- finalize()可以进行日志监控、监听重要对象被回收

- 监听系统GC事件。因为虚引用每次GC都会被回收，那么我们就可以通过虚引用来判断GC的频率，如果频率过大，内存使用可能存在问题，才导致了系统GC频繁调用

https://www.infoq.com/news/2017/03/Java-Finalize-Deprecated

什么是内存泄露和内存溢出？

代码小经验

常量字典——可读性

```
public Map<String, String> getRegisterArgs() {
    Map<String, String> registerArgs = new HashMap<>();
    registerArgs.put(TableFieldList.FIELD_NAME, mEtAccount.getText().toString());
    registerArgs.put(TableFieldList.FIELD_PASS, mEtPassword.getText().toString());
    registerArgs.put(TableFieldList.FIELD_CODE, mEtDepartmentCode.getText().toString());
    registerArgs.put( k: "username", mEtUserName.getText().toString());
    registerArgs.put(TableFieldList.FIELD_TEL, mEtPhoneNum.getText().toString());
    registerArgs.put(TableFieldList.FIELD_NAME_ISSYSTEM,  v: "2");
    registerArgs.put( k: "type", QueryAndOrderInfoUtil.CUSTOMER_USER_CODE);
    registerArgs.put( k: "roleid", QueryAndOrderInfoUtil.CUSTOMER_USER_CODE);
    return registerArgs;
}
```

2到底是什么意思?

分析代码运行路径——精简代码

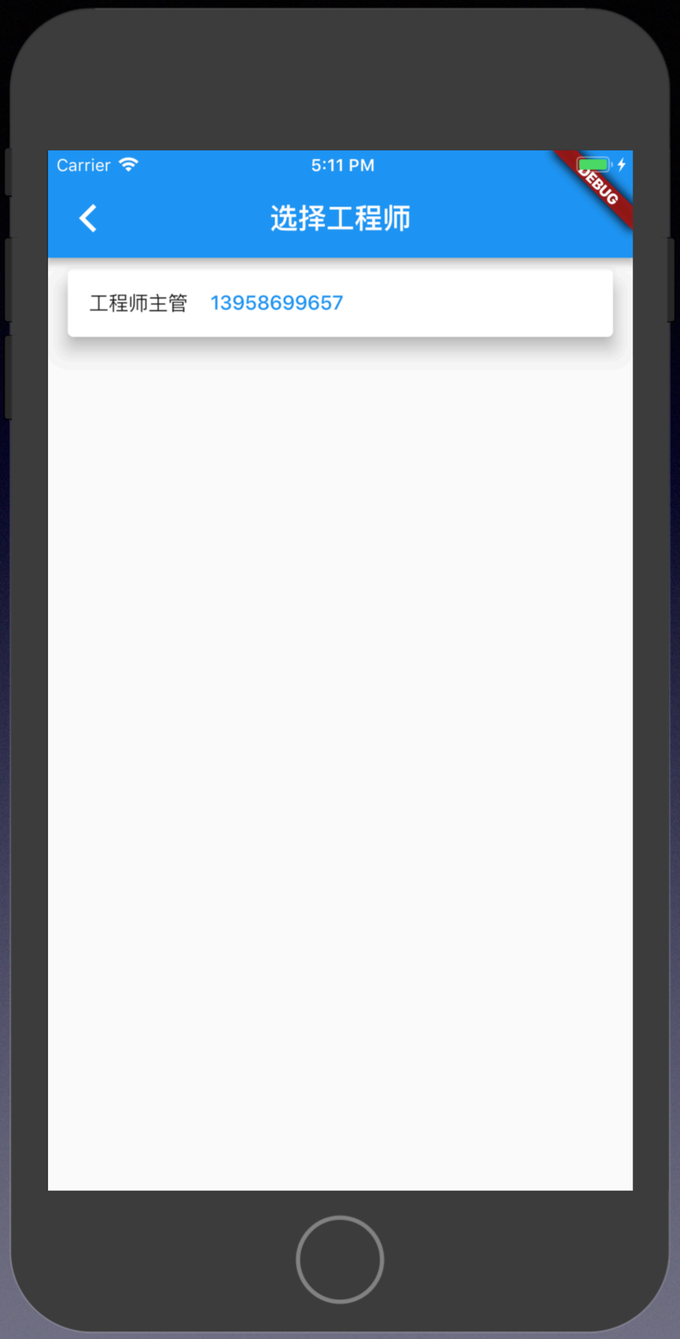
```dart
if (_currSelectedDateType != null) {
  if (_currSelectedDateType == SelectedDateType.UNIT_QUERY_START) {
    setState(() {
      QueryRepository.getInstance().unitQueryStartDate = dateStr;
    });
    _currSelectedDateType = null;
  } else if (_currSelectedDateType == SelectedDateType.UNIT_QUERY_END) {
    _currSelectedDateType = null;
  }
}
```

```
if (_currSelectedDateType != null) {
  if (_currSelectedDateType == SelectedDateType.UNIT_QUERY_START) {
    setState(() {
      QueryRepository.getInstance().unitQueryStartDate = dateStr;
    });
    _currSelectedDateType = null;
  } else if (_currSelectedDateType == SelectedDateType.UNIT_QUERY_END) {
    _currSelectedDateType = null;
  }
}
```

```
if (_currSelectedDateType != null) {
  if (_currSelectedDateType == SelectedDateType.UNIT_QUERY_START) {
    setState(() {
      QueryRepository.getInstance().unitQueryStartDate = dateStr;
    });
  } else if (_currSelectedDateType == SelectedDateType.UNIT_QUERY_END) {
  }
  _currSelectedDateType = null;
}
```

确认数据关系上关联——保证数据准确


设备位置：
0/140
是否指派工程师：
不指派
指派
指派工程师：
工程师主管
选择工程师
选择协助人
清除内容
提交
首页
单位查询
申请
我的工单
个人中心


Carrier
5:11 PM
DEBUG
选择工程师
工程师主管 13958699657


设备位置：
0/140
是否指派工程师：
不指派
指派
指派工程师：
工程师主管
选择工程师
选择协助人
清除内容
提交
首页
单位查询
申请
我的工单
个人中心

```java
class UserBean {
  int id;
  String name;
  String username;
  String pwd;
  String tel;
  String address;
  String department;
  String isSystem;
  String type;
  String sex;
  String description;
  int roleid;
  int groupId;
  String createDate;
  String modifyDate;
```

```java
String assignedEngineerName;
String assignedEngineerId;
```

```java
UserBean assignedEngineer;
```

谢谢