Building Programs & Guava Java Library

Building

1. Program construction is not always successful. What alternatives does MAKE have if one of the commands in a Makefile cannot be ended correctly? What must MAKE take into account if the target has already been partially created but the creating command is ended with an error code?
4 marks

Answer:

The way MAKE is designed, every time it executes a command it checks the exit status of that command. If it was successful with exit status as 0, then MAKE proceeds to the next command. However if the exit status returned is 1 indicating failure, then MAKE immediately stops, but this might not be what is desired.

Based on the requirements, MAKE provides different options to allow the program construction to continue. Of course, it is up to the user to determine if and what to use. In case we want to ignore generic errors while executing some commands, for example, while creating a file on execution. This might fail if the file had already been created but this is not an error which should cause the program construction from stopping.

To handle this MAKE allows us to add a "-" before the line's text. Now when the line is executed, MAKE continues with the execution of the succeeding commands even if this particular command fails. If we want to ignore errors then we can execute MAKE with the '-i" command wherein all the errors are ignored. But this kind of error handling depends on the user actually providing those details.

There might be scenarios where it might fail on lines which the user hasn't handled. If it is desired that the scripts needs to run the entire course irrespective of whether they encounter errors which are handled or not, MAKE provides the "-k" option. This lets MAKE know that it has to proceed to the end no matter what.

2. If the object files grammar.o and dictionary.o in our MAKE example are deleted, but spellcheck is kept, during the next run, MAKE recreates spellcheck and all object files again - even if the source files are unchanged.
Suggest a modification of the MAKE algorithm that does not recreate the given target if there are unchanged source files and missing intermediate files.
Are there cases where this optimization is unreasonable?
4 marks

Answer:

In MAKE, a file can basically be made by a sequence of implicit rules (Chains of Implicit Rules). For the question posed here, grammar.o and dictionary.o are referred to as intermediate files. However when the MAKE command is run again, it is stated that spellcheck is recreated along with the missing object files again. This seems like an unnecessary overkill. If there are no changes to the source files from which the initial object files were created then why recreate them in the first place

After going through the documentation on gnu.org, it is clear that intermediate files are remade using their own rules like other files. However an intermediate file is treated differently in two ways. One is, if the intermediate file does not exist, then MAKE leaves it alone, unless there is some reasons to update it. The second difference is, eve if it does create the intermediate file again, it will delete it later on when it is no longer needed. This ensures that a file which did not exist earlier will not exist again after the MAKE command finishes executing.

Therefore to achieve the result expected in the problem we can mark the grammar.o and the dictionary.o files as intermediate files by listing it as a prerequisite of the special target .INTERMEDIATE. This will ensure that these files are effectively considered as intermediate files, no matter what and therefore even if they are deleted, they won't be created again, unless their source files have changed too.

The optimization might be unreasonable in scenarios where it appears that the same implicit rule appears more than once in a chain. MAKE will not allow such a rule to stand, and will not run the linker twice. This prevents infinite loops from occurring while searching for the implicit rule chain.

3. In order to avoid the purely time-oriented change mechanism of MAKE, many developers of MAKE files use the following trick: during the reconstruction of a component A, a component called A' is created and compared with the existing A. If A' and A are different, A' is copied to A. Otherwise, A remains unchanged, include the date that it was changed.
Argue that this trick can reduce the time taken to build a target.
With this kind of trick, can the steps that will be necessary (for building a target) be determined before the actual construction of the program?
Argue that a MAKE that follows the rules for reconstruction given by the Rebuild Theorem in the class notes (slide 5) will nevertheless recreate the components that depend on A. What must MAKE take into account so that this trick works?
6 marks

Answer:

The trick explained in this problem will definitely reduce the time taken to build a target. Here's why. In a normal Make process, when the utility is run, it examines the modification times of the files and determines what needs to be regenerated. Basically files that are older than files they depend on must be regenerated. However regenerating one file may cause others to become old and this ends up making several files regenerate unnecessarily. As a result we can notice that the time consumed by a normal Make process will be high depending on how many files it ends up regenerating etc., Now on the other hand, when the trick mentioned in the problem is applied, this is how it would reduce the time. When the component A' is created from A, there is no compilation happening right away. Now when we compare the files we can always make out if the files are different. If they are, then it helps us decide on the compilation of that particular unit. But we can avoid the disadvantages of the original Make over here. Instead of ending up regenerating all the files associated with component A, we can regenerate only the ones that have changed. This would serve as a huge time saver if this trick is used.

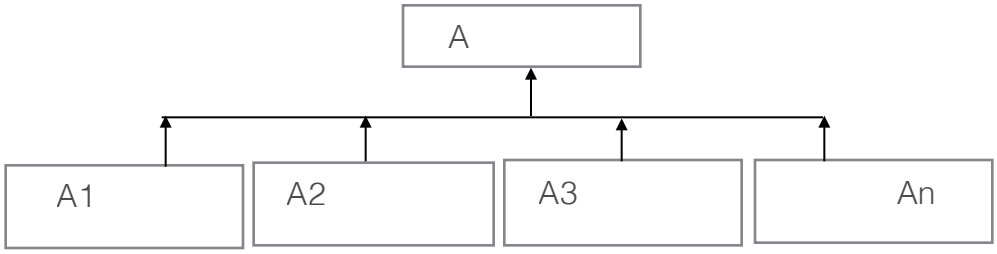
To answer the second part of the question if the necessary steps for building a target can be determined before actual construction of the program. I definitely think it is possible using this trick. During the initial makefile creation, we would end up setting up all the required

dependencies for the component A. Since the component A is copied into A', we can compare all the dependent files too during this process. Only if a difference is found will the actual re-construction of A start else it will be ignored.

The rules of reconstruction specified by the Rebuild Theorem has the limitation that the derived component A has to be rebuilt either if A does not exist or if at least one Ai from A1, A2…An has changed or needs to be rebuilt. This set of core constraints implies that no matter what changes A will end up getting rebuilt. When that happens, the components depending on it also will get rebuilt because their timestamps would differ from that of A indicating to Make process that they need to be updated too. For this trick to work, MAKE must basically ensure to check for differences between the original value of A and the current one. In addition to this, it has to verify if any of the components which depend on A have changed too.

4. Sketch out a design for a tool similar to MAKE which detects changes not by means of file timestamps, but by means of the actual content of the components. Specifically, for each derived component A, you must remember the content of all components A1,...,An, on which A depends. A should be reconstructed iff there are differences in the contents of A1,...,An. How would you save disk space? How can you get MAKE to focus on the non-commented content of A1,...,An?
4 marks

Answer:



Dependency

A: A1, A2….An

The tool will contain the list of dependencies that exist between the derived component A and the other components. Now whenever we run a command for Build, we can do a diff of the file A along with the other files which depend on it. If there are differences in any of the files, then we can rebuild that file alone. Since this approach depends on using the diff of the files to determine if a rebuild is required or not, we can have tremendous improvements in performance.

To save disk space, we can follow the Google Approach, wherein we will build only one set of object files for a class. Any class that needs that class will link to this object. As a result the number of duplicate objects created would be literally zero. The other advantage of following this approach is that we can always ensure that when one of the files that others depend on changes, all the components A1, A2…An etc see it immediately and don't have to actually rebuild everything again.

In order to ensure MAKE focuses only on the non-commented contents of A1, A2….An, we can embed commands inside the MAKE file to filter those comments in the first place. Following the rebuild theorem approach, since we create a temporary file A' prior to comparing, we can do a similar thing here. We can create a temporary file where the file is parsed to remove the comments. Since we already do a diff before creating a temporary file, it is easy for us to determine if the file has changed or not. If there are no changes, then no temporary file needs to be generated else we generate the temporary file. Using this file, we can perform the object creation and linking to the original files. This would also give us a slight reduction in the disk space too. This way MAKE will focus only on the non-commented contents of the files.

5. List three ways in which Apache Ant is qualitatively different from MAKE.
3 marks

Answer:

The three ways in which Apache Ant is different from MAKE is as given below:

i) Ant is extended using Java classes instead of being extended with shell-based commands. In Ant the configuration files are XML-based target trees where various tasks get executed
ii) Ant gives us the ability to be cross-platform and work anywhere, everywhere unlike Make which limits us to the OS or the OS type we are working on
iii) Since the Ant files are XML-based, they don't have to deal with problems like "tab problem" encountered in Make.
iv) It is also easy to invoke Ant from the command line or integrate with various IDEs which may not really be possible with Make

6. Watch this Google TechTalk on building at very large scale and, in 2-4 sentences each, describe
Three things you learned
One thing you think might be useful in your current dev environment
One thing that you think will not be useful in your current dev environment
10 marks

Answer:

The Google TechTalk on building very large scale was very informative. The concepts used to achieve the performance desired, even though they seem simple and easy was an eye opener. Below are the three things that I learnt from the talk:

i) The concept of "Forward Graph" was very interesting. It was particularly interesting to see how it helps avoid duplicating work. Also the fact that it helps avoid compiling the same source code repeatedly as part of different builds and uses a common cache to achieve it was a very nice concept to learn
ii) The fact that caching is used extensively to achieve boosts in performance of the build process and how it is done was also very informative
iii) The concept of "Dirty" graph wherein only the pieces of code that have changed are compiled and then cached for use by other compilation processes etc., is a nice feature, especially since the graphs that google might need will only grow larger

iv) Finally the fact that cloud is used to execute the build using dedicated m/c's and the output is made available as fast as possible is a very cool thing to have. Also the fact that some actions that you need might already have been done and cached is a huge boost when it comes to time constraints

The one thing that might be useful in my current dev environment is:

In our work environment, we definitely have some duplication of work. If code is changed in one project whose jars are used in other projects, we have to go in and update those dependencies. If we could use the concept of the Forward graph, we can definitely save a lot of time. For instance, we can set up dependencies or the Action graph which indicates that Project A and B depend on the JAR from Project C. Now if any changes are made to Project C and a new JAR is updated, it can automatically run the actions for updating the Project A and B also at real time.

This way the manual intervention required would be redundant resulting in much more stable build and release cycles.

The one thing that will not be useful in my current dev environment is:

In our dev environment, we don't have as much amount of code as the Google build system deals with. Therefore, sending the build processes to cloud and having dedicated machines etc., will be an overkill. So this is definitely something that won't be used in my current dev environment. Even though conceptually the Google build system is very performance oriented and stable, it might just be overdoing it for our environment

Guava

Short (2 marks each)

Multiplying integers can lead to overflow. Suggest a way in which you can detect and handle an overflow. (In Java 1.8, Math.multuplyExact() will solve this problem, but you should use Guava.)

Answer:

Using the guava method "public static int checkedMultiply(int a, int b)", we can detect and handle an overflow. It returns the product of a and b if it does not overflow otherwise it would thrown an "ArithmeticException" in case of overflows.

Write a one-line check that an integer variable i is a valid index into an array A. If not, throw a IndexOutOfBoundsException with a message indicating i and A's size.

Answer:

Either "public static int checkElementIndex(int index, int size)" or "public static int checkPositionIndex(int index, int size)" can be used.

The first one verifies if the index specifies a valid element in an array A and the second one verifies if the index specifies a valid position in an array. In both the cases, if it the index fails to match, then an IndexOutOfBoundsException is thrown

Let's say you need store key-value pairs, and in addition to the key to value mapping, you need a value to key mapping, e.g., if you want to go from state to capital, and vice versa. One approach to map values back to keys is to maintain two separate maps and keep them both in sync. Explain what could go wrong, and how Guava can be used to solved the issues you came up with.

Answer:

The problem with the first approach of having two separate maps is that they have to be kept in sync and there is no guarantee that there is unique bidirectional relationship between the two. For example, the same capital name may be associated with two states causing issues when we try to access the State using a Capital name.

We can solve this issue by using the Guava Bimap. The guava bimap also referred to as "bidirectional map" is a map that preserves the uniqueness of its values as well as that of its keys. This basically enables the bimaps to support an "inverse view" (as specified in the google java docs). Therefore using this, we can have another bimap which contains the same attributes as the original ones but with reversed keys and values.

One way to compute the intersection of two Set objects in Java is to iterate over the first set and use contains() on the second. Can you find a better way using Guava?

Answer:

Guava provides the method "public static <E> Sets.SetView<E> intersection(Set<E> set1, Set<?> set2)" in order to handle the intersection functionality.

This method returns an unmodifiable view of the intersection of two sets. The returns set contains all the elements that are contained by both the backing sets. Therefore by using this method we can get the intersection of two sets without having to iterate through the first set and then using contains() on the other.

For a primitive array, you need to iterate over the array to test if a value is present. Can you perform the contains() in better way with Guava?

Answer:

In guava, in order to verify if an array contains a value, we can just use the function

"boolean contains(prim[] array, prim target)".

This method will determine if the specified element exists in the specified array or not and return a boolean result indicating the same.

From the Javadoc for String determine what ",a,,b,".split(",") returns. (Do not execute the code, you must base your answer on the Javadoc.) Write a one line of Guava that will return [null, "a", null, "b", null] for such a string.

Answer:

On splitting the String **“,a,,b,”.split(“,”)** we would get an empty value for the 0th position and “a” for the second position. This happens because the split happens on the first appearance of “,” even if there is nothing preceding it. However this behavior may not be consistent because of the quirky behavior of the built in Java utilities.

In case of Guava, we can do a split using the method

Splitter.on(',').trimResults().omitEmptyStrings().split(",a,,b,");

Why is there no Pair<T,U>, Triple<A,B,C>, etc. functionality in Guava? (It’s not in the JDK either.)

Answer:

A class Pair or Triple doesn’t really indicate any semantics about the relationship between the values. We don’t know what the relationship is between T and U, in case of Pair (or) the relationship between A, B and C in case of a Triple. Hunter Gratzner gives a good argument on the why a Pair should not be added in Java in the link here.

More involved

Let’s say you want to maintain a set of strings. The number of elements in the set is very large. All you are doing with the set, after it’s been built, is lookups. You have some freedom with respect to accuracy, specifically, you can occasionally return true, even if the query string is not present, but cannot return false if it is present. One approach is to store hashcodes for the strings. Find a more memory-efficient data structure for this purpose. (5 marks)

Answer:

In order to solve this problem we can use the concept of Bloom Filter. Bloom Filter provides a probabilistic data structure which is designed to be rapid and memory efficient, and indicates whether an element is present in a set or not. Of course the efficiency gained is because it is a probabilistic data structure, which can only say whether an element either definitely is not in the set or maybe in the set.

So we can use this concept to solve our problem. Once we build the large set, we can add an element to the bloom filter. When an element is added, it is basically hashed a few times and then we set the bits in the bit vector at the index of the hashes to 1.

Now when we need to do the lookup, we hash the string with the same hash functions we used while adding the element. After this, we can check if those values are set in the bit vector. If the bits are not set, then it indicates that the element definitely does not exist in the set. On the other hand, if the bits are set, then it can be inferred that the value maybe exists in the set. The reason this is “maybe” is because there is definitely a possibility that some other combination of elements might also have set the same bits.

Therefore by using the concept of BloomFilter provided in Guava

BloomFilter<T> with create(Funnel funnel, int expectedInsertions, double falsePositiveProbability),

we can achieve the intended behavior as outlined in the problem. This approach is definitely the most efficient way of handling large sets for lookup purposes

Suppose you are writing a program that operates on images. Images are specified by a URL; its contents are at a remote server. (Think of the contents as a base64 string.) To boost performance, you would like to cache the image contents in your program's RAM. What are the key features would you like the cache to have and the challenges you'd encounter in implementing it? Write a one-line assignment statement that creates such a cache for your program that incorporates these features. (5 marks)

Answer:

The key features that are needed in a cache can be summarized as:

i) Improvement in speed of data retrieval
ii) Avoid duplication of values stored in the cache
iii) Have the flexibility to remove values from cache when they are not needed anymore
iv) Most important of all, provide easy access to the data stored in the cache
v) Capable of having a set value so that it does not exceed the capacity of a RAM.

The challenges we face while implementing this is:

i) In case we use the Java Map for strong the values,we are responsible for eviction of entries too in order to keep the size to a given limit.
ii) For multithreaded environments, a simple java Map is not sufficient and we have to use ConcurrentHashMap for solving the concurrency problems. However this causes the code to get complicated and also we have to deal with the fact that keys can be added multiple times from different threads concurrently

To solve all these issues, we can use the Cache implementation provided by Guava. Using CacheLoader we can create a new Cache and set its size to some value too, if desired.

```
LoadingCache<String, String> imageUrlCache =
CacheBuilder.newBuilder().maximumSize(1000).build(new CacheLoader<String, String>() {
                                        public String load(String imgKey) throws AnyException {
                                                                return imgKey.toUpperCase();
                                                                }
                                                });
```

The sample above from Guava website clearly shows how a cache can be built using the CacheBuilder and CacheLoader. We can create a Cache with the image name as the Key and the URL to the image as the value. Once the cache is created using Guava, we can retrieve the image from the source by referencing the URL stored in the cache.