Concurrent Build System in Go

Duarte Pereira nº55409
Rafael Costa nº45464
PCLT 2022/2023
FCT/UNL

Graph

Our program starts by building a graph, where the nodes represent one compilation unit, and the edges represent a dependency relation between the units. This graph is implemented as a map where the keys are strings (the unit name), and the values are lists of nodes (the compilation units that that unit depends on). We also found it useful to have an auxiliary data structure that maps the compilation unit's names to a pointer to their node. Both of these data structures are built in the makeGraph function, which loops through all the dependency relations, and creates nodes for both the target and the dependencies. Every dependency node gets added to the target node's list. Our graph building function does not differentiate between internal nodes and leaves, because it generates nodes both for the target and for the dependencies of each rule, and adds even the dependencies as entries to the graph. However, this means that we have to check for repetitions every time we add a new node to the graph, as the target of one dependency can be the dependency of a different target (that is, we have to be careful to, instead of creating the same new node repeatedly, getting the one that already exists and inserting it in the new location of the graph).

Nodes

Each node stores a channel with which it communicates with its "parent" nodes, and a list of channels from which it receives messages from its "children". It also stores the number of parents that that node has, that is, the number of files that depend on that node. This is because twhen the node finishes building, it needs to know how many times to send the message to its parents.

Channels

The channel fields are populated in the addChannelsToNodes function after the building of the graph, which for every dependency target, loops through all of its children. If that child node still doesn't have a channel to its parents, it is created, and the child's channel to its parents is added as one of the target node's channels to its children. If the node already had a channel to its parents, this means that the node currently being processed is not its first parent, therefore, the number of parents of that node needs to be incremented.

Building the Nodes

After this is done, a goroutine is launched for each node in the graph, that attempts to build that particular node. This function starts by waiting for messages from all of the node's children, which signal that the child node has built successfuly. This, assuming that the graph is well-built, ensures that each dependency target is only built after all of its dependencies are built, which should guarantee the correctess of our solution.

But our program does not build all nodes all the time, only if they're out of date (were last built before any of their dependencies). The first step to ensure this is to receive from the child nodes the timestamp of their most recent build, and store the most recent one. Then, if the object file does not exist, it is always built. If the node is a leaf and the object file exists, it is not built. If the node is internal or the root, and the object file already exists, we need to analyse the timestamps. If the timestamp of the object file is more recent than the timestamp received from the children, then the file is up-to-date and nothing needs to be done. Otherwise, if the timestamp of the object file is older than that of the children, it needs to be rebuilt.

Whether the node is built or not, it always sends a message to its parents. If it built, it sends a success message with the current timestamp. If it did not build because it was up to date, it sends a success message with the timestamp of the last update of the file (the last time the file was built). If the build fails for some reason, an error message is sent with a timestamp of time.Time{} (equivalent to 0 time). The node sends one message to each parent, using the ParentNum field explained previously.

This process of sending the timestamps of the last build to the parents makes it so that only necessary builds are triggered, and nodes that are already built aren't updated needlessly.

Data Races

Since each node is built in its own goroutine, and the goroutines do not write either on shared variables or on other nodes, and messages over channels are synchronous, no data races on program variables are possible. Data races are also not possible on the object files, since each one is generated by its node's goroutine, and properties of that file are also only accessed by that goroutine.

To validate our program we tested it using different dependency files, which we placed in the test_files directory. We also ran it multiple times with the -race flag, which detects data races, and none was detected.